НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Кафедра обчислювальної техніки

«На правах рукопису»
УДК __004.052.42___

До захисту допущено:

Завідувач кафедри

_____ *Сергій СТІРЕНКО*
«___»_____2021 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-науковою програмою «Комп'ютерні системи та мережі»

зі спеціальності 123 «Комп'ютерна інженерія»

на тему: «Метод та засоби тестування криптографічних алгоритмів на основі булевих перетворень»

Виконала:
студентка VI курсу, групи ІВ-91мн
Дорошенко Анна Юріївна _____

Керівник:
доцент, к.т.н., доцент,
Марковський Олександр Петрович _____

Консультант з нормоконтролю:
професор, д.т.н., професор,
Кулаков Юрій Олексійович _____

Рецензент:
декан ФПМ, д.т.н, професор,
Дичка Іван Андрійович _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2021 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет (інститут) *Інформатики та обчислювальної техніки*
(повна назва)

Кафедра *Обчислювальної техніки*
(повна назва)

Рівень вищої освіти – другий (магістерський) за освітньо-науковою програмою
Спеціальність *123. Комп'ютерна інженерія*
(код і назва)

Спеціалізація *123. Комп'ютерні системи та мережі*
(код і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри
Стіренко С.Г.
(підпис) (ініціали, прізвище)
« » 20***21*** р.

ЗАВДАННЯ
на магістерську дисертацію студенту
Дорошенко Анні Юріївні
(прізвище, ім'я, по батькові)

1. Тема дисертації Метод та засоби тестування криптографічних алгоритмів на основі булевих перетворень

Науковий керівник дисертації доц.,к.т.н., доц. Марковський О.П.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «12» березня *2021* р. № 809-с

2. Строк подання студентом дисертації *26 квітня 2021 р*

3. Об'єкт дослідження *криптографічні алгоритми, основою яких є нелінійні булеві перетворення.*

4. Предмет дослідження *методи прискореного тестування нелінійності булевих перетворень.*

5. Перелік завдань, які потрібно розробити: *виконання аналітичного огляду методів тестування криптографічних алгоритмів, аналітичний аналіз та порівняння існуючих методів прискорення визначення нелінійності булевих перетворень, розробка, теоретичне обґрунтування та дослідження нового*

методу підвищення ефективності тестування криптографічних алгоритмів на основі булевих перетворень, розробка програми на мові Rust для дослідження розробленого методу шляхом експериментального моделювання, порівняння розробленого методу з існуючими методами визначення нелінійності та оцінка ефективності запропонованого методу.

6. Консультанти розділів дисертації:

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Кулаков Ю.А., професор		

7. Дата видачі завдання 26.11.2020

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Строк виконання етапів дисертації	Примітка
1.	*Затвердження теми роботи*	*10.12.2020-15.12.2020*	
2.	*Вивчення та аналіз завдання*	*15.12.2020-31.01.2021*	
3.	*Розробка архітектури та загальної структури системи*	*01.02.2021-10.02.2021*	
4.	*Розробка структур окремих підсистем*	*10.02.2021-20.02.2021*	
5.	*Програмна реалізація системи*	*20.02.2021-10.04.2021*	
6.	*Оформлення пояснювальної записки*	*10.04.2021-25.04.2021*	
7.	*Передзахист*	*26.04.2021*	
8.	*Захист*	*17.05.2021*	

Студент _____ А.Ю. Дорошенко
(підпис) (ініціали, прізвище)

Науковий керівник дисертації _____ О.П. Марковський
(підпис) (ініціали, прізвище)

РЕФЕРАТ

на магістерську дисертацію

виконану на тему: Метод та засоби тестування криптографічних алгоритмів на основі булевих перетворень

студенткою: Дорошенко Анною Юріївною

Робота складається із вступу та 4 розділів. Загальний об’єм роботи: 81 аркушів основного тексту, 4 ілюстрації, 5 таблиць, додатки. Для виконання магістерської дисертації було використано інформацію з 54 літературного джерела.

Актуальність. Розвиток інформаційної інтеграції дозволив вивести обробку великих об’ємів інформації на якісно новий вищий рівень. Як наслідок, ці можливості сприяли загостренню проблеми захисту даних.

Широкий клас сучасних механізмів захисту інформації базується на використанні криптографічних методів, функціональною основою яких є незворотні булеві перетворення. У свою чергу, оцінка нелінійності булевих перетворень є важливою складовою тестування стійкості до зламів алгоритмів захисту інформації.

Одним із очевидних способів підвищення криптостійкості алгоритмів є використання булевих перетворень від більшої кількості змінних. Це призводить до ускладнення процесу їхнього тестування, оскільки виникає багатократне збільшення потрібного на це часу.

Зазначені вище чинники роблять задачу створення нових методів прискореного тестування сучасних криптографічних алгоритмів на основі незворотних булевих перетворень актуальною та нагальною на сьогоднішній день.

Мета і завдання дослідження. Метою магістерської дисертації є підвищення ефективності тестування криптостійкості до зламів алгоритмів захисту інформації, що мають за основу булеві перетворення, шляхом прискорення процедури визначення їх нелінійності.

Для досягнення поставленої мети було поставлено та вирішено такі завдання:

- виконання огляду методів тестування криптографічних алгоритмів.
- аналітичний аналіз та порівняння існуючих методів прискорення визначення нелінійності булевих перетворень.
- розробка, теоретичне обґрунтування та дослідження нового методу підвищення ефективності тестування криптографічних алгоритмів на основі булевих перетворень.
- розробка програми на мові Rust для дослідження розробленого методу шляхом експериментального моделювання.
- порівняння розробленого методу з існуючими методами визначення нелінійності та оцінка ефективності запропонованого методу.

Об'єкт дослідження – криптографічні алгоритми, основою яких є нелінійні булеві перетворення.

Предмет дослідження – методи прискореного тестування нелінійності булевих перетворень.

Методи досліджень базуються на основних положеннях теорії булевих функцій, криптографії, теорії ймовірностей, теорії оптимізації та динамічного програмування, основні положення статистичного моделювання. Для експериментального дослідження використовувалися методи комп'ютерного моделювання.

Наукова новизна одержаних результатів роботи полягає у наступному:

Теоретично обґрунтовано, розроблено та досліджено метод прискореного визначення нелінійності булевих перетворень, який відрізняється від відомих використанням динамічного програмування для побудови лінійних апроксимацій, за рахунок чого досягнуто прискорення визначення нелінійності булевих перетворень від великої кількості змінних.

Особистий внесок здобувача полягає в теоретичному обґрунтуванні одержаних результатів, їх експериментальній перевірці та дослідженні, а також у

створенні програмних продуктів для практичного використання одержаних результатів.

Практична цінність отриманих в магістерській дисертації результатів полягає в тому, що вони дозволяють прискорити процес тестування криптостійкості криптографічних алгоритмів та збільшити надійність оцінок здатності протистояти лінійному криптоаналізу.

Апробація результатів дисертації

Основні результати дисертації доповідались та обговорювались на 3-х міжнародних науково-технічних конференціях:

1. Міжнародна наукова конференція “Security, Fault Tolerance, Intelligence: ICSFTI2019”. м.Київ, 14-15 травня 2019 р.

2. ІІ-га Міжнародна науково-практична конференція “Наука та концепції”. м.Київ, 29-30 квітня 2019 р.

3. Міжнародна наукова конференція “Security, Fault Tolerance, Intelligence: ICSFTI2020”. м.Київ, 13 травня – 15 липня 2020 р.

Публікації

Основні положення магістерської дисертації опубліковані в 4 наукових працях, серед яких три – матеріали наукових конференцій та одна – стаття у фаховому журналі.

1. Doroshenko A. Acceleration of boolean transformations nonlinearity testing for cryptographic algorithms / Anna Doroshenko, Oleksandr Markovskyi // International Conference ICSFTI2019 (Kyiv, May 14−15, 2019). Kyiv, 2019. – P. 35-40.

2. Rusanova O. Energy-aware task scheduling algorithm for mobile computing / Olga Rusanova, Igor Boyarshin, Anna Doroshenko // International Conference ICSFTI2020 (Kyiv, May 13, June 15, 20120). Kyiv, 2020. – P. 107-113.

3. Дорошенко А. Ю. Метод прискореного тестування нелінійності булевих перетворень криптографічних алгоритмів / А.Ю. Дорошенко, В.Ю. Куц // Матеріали

II міжнарод. наук.-практ. конф. Наука та концепції: (м. Київ, 29-30 квіт. 2019 р.). Київ, 2019. – С. 15-18.

4. Boyarshin I. Request balancing method for increasing their processing efficiency with information duplication in a distributed data storage system / I. Boyarshin, A. Doroshenko, P. Rehida // Technical sciences and technologies. – 2021. – № 2 (26).

Ключові слова

Лінійний криптоаналіз, нелінійність булевих перетворень, шифроблоки, s-блок, тестування криптографічних алгоритмів, оцінка криптостійкості.

ЗМІСТ

ВСТУП

На сьогоднішній день все більшим фактором у будь-якій сфері людської діяльності стає інформаційна інтеграція, яка дозволяє обробляти значну кількість різноманітних даних за відносно незначний проміжок часу. Як наслідок, по мірі збільшення інтенсивності цієї інтеграції з'являється нагальна потреба адаптувати існуючі та винайти нові засоби для того, аби забезпечити належний рівень захисту інформації.

Більшість існуючих механізмів та систем для захисту інформації базуються на основі виконання певних алгоритмів із криптографічною складовою, значна частина яких реалізується за допомогою булевих перетворень з властивістю незворотності. Для того, аби мати можливість якісно оцінити криптографічну захищеності подібних алгоритмів захисту даних, для булевих перетворень зазвичай використовується показник їхньої нелінійності.

Головною проблемою при знаходженні нелінійності булевого перетворення є її значна обчислювальна складність та, як наслідок, значний обсяг часу, необхідний на її виконання. З метою запобігання зламу криптографічних алгоритмів розповсюдженою практикою є збільшення кількості змінних у булевому перетворенні, що призводить до експоненційного підвищення кількості необхідних обчислень.

Саме тому питання створення нових способів прискореного знаходження нелінійності криптографічних алгоритмів захисту інформації на основі булевих перетворень з властивістю незворотності є актуальним для сучасного етапу розвитку комп'ютерних систем та мереж.

РОЗДІЛ 1

АНАЛІТИЧНИЙ ОГЛЯД МЕТОДІВ ОЦІНКИ СТІЙКОСТІ ДО АТАК АЛГОРИТМІВ КРИПТОГРАФІЧНОГО ЗАХИСТУ

1.1. Огляд криптографічних алгоритмів на основі булевих перетворень

Можна виділити такі основні завдання захисту інформації, які вирішуються в сучасних комп'ютерних системах та мережах:

- захист даних, що розміщені в пам'яті вбудованих комп'ютерних систем, та повідомлень, що пересилаються користувачами комп'ютерних мереж, від незаконного доступу;
- захист інформації, що розміщена на носіях різних типів вбудованих комп'ютерних систем, та повідомлень, що пересилаються мережею, від незаконної зміни, тобто гарантування цілісності та оригінальності даних;
- захист інформаційних та обчислювальних ресурсів вбудованих комп'ютерних систем від незаконного доступу нерозпізнаними користувачами.

Зазначені вище задачі вирішуються організаційними, алгоритмічними, технічними та програмними засобами, які ефективно доповнюють один одного, створюючи нероздільну систему. Слід виділити такий елемент цих систем, який широко використовується на практиці для вирішення описаних задач захисту даних у комп'ютерних системах та мережах, як спеціалізовані алгоритми. Алгоритми мають істотну перевагу перед технічними засобами захисту інформації у вигляді їхньої меншої вартості, а також різнопланового використання у функціональному плані [2, 3].

Аби проаналізувати, порівняти та оцінити алгоритми захисту інформації необхідно виділити групу основних критеріїв для визначення якості цих

алгоритмів. Проте завдання визначення критеріїв не є однозначним, оскільки неможливо виділити універсальні критерії, які б задовольнили оцінку усіх класів алгоритмів. Для оцінки якості алгоритмів захисту інформації різних класів виділяють власні критерії, спираючись на специфічні умови застосування конкретного класу алгоритмів. Так, значущі критерії якості для одного класу алгоритмів є зовсім неістотними для іншого класу. Наприклад, суттєвим критерієм оцінки якості алгоритмів «з відкритим ключем» є час, за який формуються закриті та відкриті ключі, проте для інших класів алгоритмів такий критерій є несуттєвим. Узагальнити критерії якості оцінки алгоритмів захисту інформації різних класів можна таким чином [21]:

- ступінь захищеності інформації, що є пропорційним обчислювальним ресурсам, які необхідно було використати для зламу конкретного алгоритму захисту даних;
- швидкість роботи алгоритму на процесорі з універсальною архітектурою;
- стійкість даних, закодованих за допомогою алгоритму захисту, до зумисних та незумисних завад при збереженні інформації на носії різних типів та при передачі даних по каналам комп’ютерних мереж;
- потенціальна апаратна реалізація алгоритму захисту даних, яка забезпечить максимально ефективну роботу алгоритму, на програмованих та спеціалізованих надвеликих інтегральних схемах.

Важливо зазначити, що значення вказаних критеріїв мають обернену залежність: при покращенні одного показника найімовірніше будуть погіршуватися інші. Тому оцінка якості конкретного алгоритму захисту інформації за вказаними критеріями є компромісним показником, який показує, наскільки гарно було визначено значущість кожного з критеріїв, зважаючи на область застосування алгоритму. Можна виділити такі принципи визначення значущості критеріїв, які

складають оцінку якості алгоритмів захисту інформації в комп’ютерних системах та мережах:

1. Алгоритм повинен забезпечувати такий ступінь захисту даних, за якого прибуток від реалізації зламу криптографічного алгоритму буде менше вартості обчислюваних ресурсів, які знадобилися для його зламу. Тобто отримання доступу до захищених даних є економічно невигідним.
2. Вартість реалізації криптографічного алгоритму, що складається з вартості використаних обчислювальних ресурсів та супровідних витрат, повинна бути менше за втрати, які принесе здійснення незаконного доступу до захищених даних. Тобто захист конкретних даних повинен бути економічно раціональним та вигідним.

1.2. Огляд аналітичних методів порушення захисту криптографічних алгоритмів. Лінійний криптоаналіз

Доречним кроком є введення класифікації алгоритмів захисту даних та виділення класу алгоритмів, вдосконалення тестування яких виконується в цій дисертації, аби проаналізувати обчислювальні методи, використані для реалізації криптографічних алгоритмів захисту даних в комп’ютерних системах та мережах, а також дати оцінку ступені захищеності інформації, яку забезпечують ці алгоритми.

Класифікацію криптографічних алгоритмів, які використовуються в комп’ютерних системах та мережах можна проводити за різними властивостями [4].

В залежності від криптографічного перетворення, що виконує алгоритм криптографічного шифрування інформації, алгоритми можна поділити на зворотні та незворотні. Для зворотних алгоритмів є визначеними процедури прямого та зворотного перетворення. У свою чергу, для незворотних алгоритмів визначеною є тільки процедура прямого перетворення, результатом якої є отримання цифрової

сигнатури повідомлення. Процедура ж отримання зворотного перетворення є нерозв'язною задачею і може бути виконана тільки здійсненням повного перебору.

Узагальнена класифікація сучасних алгоритмічних криптографічних засобів захисту даних у комп'ютерних системах та мережах наведено на рис. 1.1.

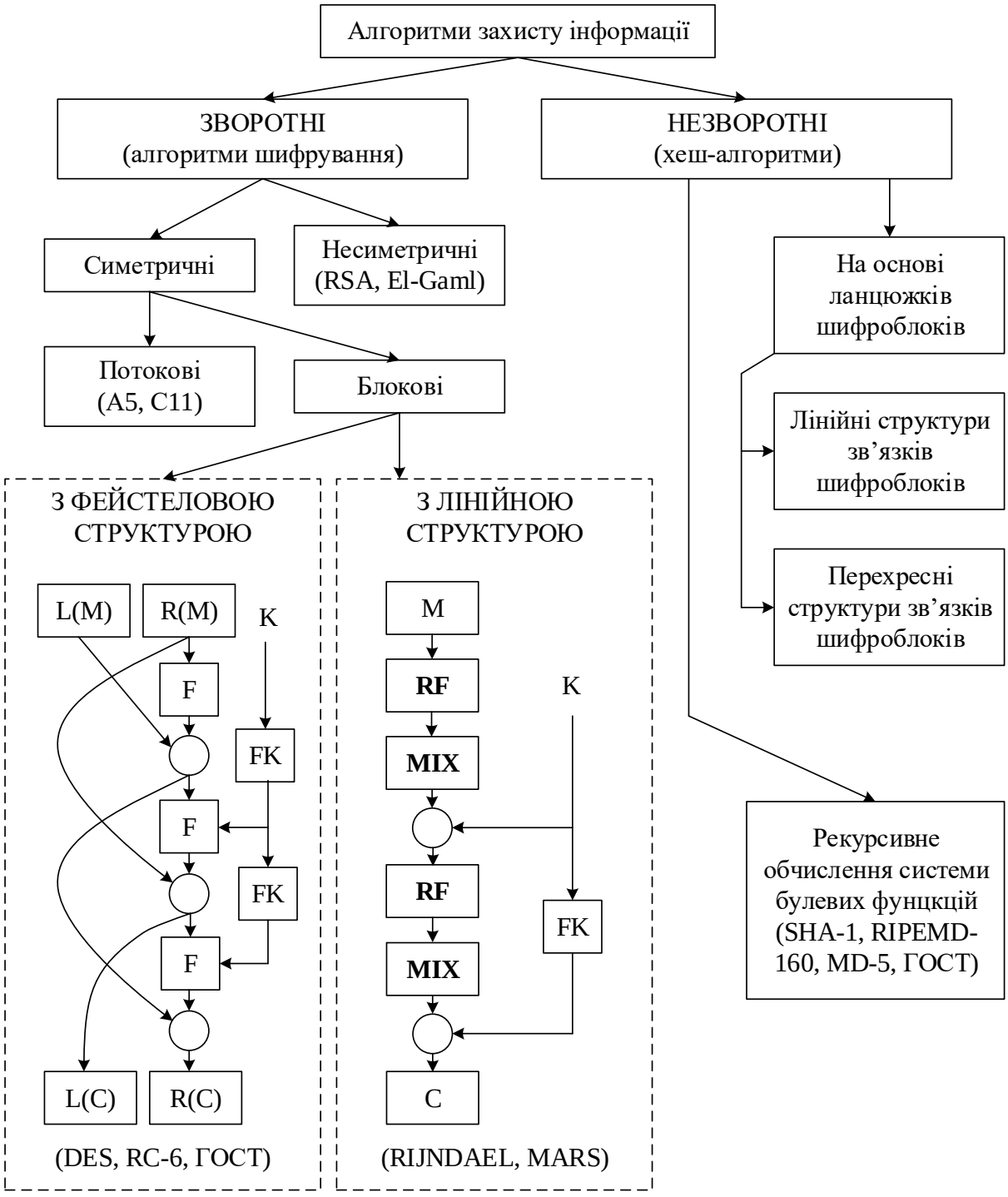

Рис. 1.1. Класифікація криптографічних алгоритмів захисту даних

Серед зворотних алгоритмів можна виділити дві групи:

– симетричні, у яких для виконання прямого та зворотного перетворення застосовуються одні й ті самі ключі;
– несиметричні («з відкритим ключем»), у яких для виконання прямого та зворотного перетворення застосовуються різні ключі. Ці алгоритми ще носять назву алгоритмів «з відкритим ключем», оскільки за протоколом один з ключів, які є різними та використовуються для шифрування і дешифрування даних відповідно, може бути відкритим.

Алгоритми симетричного типу розділяють на:

– потокові, основою яких є концепція, запропонована в середині XIX століття Клодом Шенноном: повідомлення (*message*) шифрується методом знаходження порозрядної суми за модулем 2 його коду з псевдовипадковою бінарною послідовністю, результатом чого є отримання шифроповідомлення (*cipher*). Зі сказаного випливає, що генератор псевдовипадкових бінарних послідовностей, які формуються синхронно як на приймачі, так і на передавачі, є основним функціональним модулем цього виду алгоритмів. При цьому шифрування інформаційного повідомлення виконується без розбиття його на частини, а одним потоком методом знаходження суми по модулю 2, результатом чого є висока швидкість процесу шифрування інформації.
– блокові, за алгоритмом роботи яких повідомлення ділиться на частини (блоки), що мають фіксовану довжину та шифруються окремо. Водночас для шифрування повідомлення може бути використана концепція Клода Шеннона: отримання шифроповідомлення виконується знаходженням суми за модулем 2 окремої частини повідомлення з результатом декотрого функціонального перетворення $\Theta(message, key)$, виконаного над кодом ключа. Логічно, що для дешифрування інформаційного повідомлення за

принципом Клода Шеннона, необхідно повторно порозрядно додати за модулем 2 код шифроповідомлення та результат функціонального перетворення $\Theta(message, key)$, виконаного над кодом ключа:

$$cipher = message \oplus \Theta(message, key)$$

$$message = cipher \oplus \Theta(message, key)$$

З описаного вище та формул випливає, що оберненість алгоритмів, які базуються на концепції Клода Шеннона, прямо зумовлена оберненістю операції суми за модулем 2. Розглянуті алгоритми називають ще лінійними. Така назва зумовлена тим, що текст вхідного інформаційного повідомлення переходить до шифроповідомлення лінійним чином. До цього класу алгоритмів входить більша частина симетричних криптографічних алгоритмів (наприклад, таких, як DES, RC-6, IDEA, SAFER, ГОСТ 28147-89) [28].

Основою нелінійних симетричних алгоритмів є зворотна функція $\varphi(key)$, за якої виконується: $cipher = \varphi(key, message)$ та $message = \varphi(key, message)$. При цьому як функції $\varphi(key)$ застосовують перетворення зворотного симетричного виду на кінцевих полях Галуа *GF(2n)*. Найтиповішим алгоритмом, що відносять до класу нелінійних симетричних, є Rijndael, головною ідеєю якого є виконання функціонального перетворення у рамках кінцевого поля Галуа, яке залежить від ключа. Передбачається, що таке перетворення виконується багато разів. Для того, аби відновити первинний текст, таке функціональне перетворення необхідно зробити навпаки, тобто в інверсному порядку.

Більшість алгоритмів такого класу виконують оброблення даних, розділяючи їх на частини, що називають блоками та мають конкретну довжину, тобто складаються з блокової архітектури [44]. В алгоритмах опрацювання частини повідомлення може залежати від попередніх відносно неї частин або не залежати. За цим критерієм алгоритми поділяють на залежні (які ще носять назву синхронізованих) та незалежні (які ще називають несинхронізованими). Відмітимо,

що несинхронізоване опрацювання частин інформаційного повідомлення має вагомий недолік: однакові вхідні частини даних будуть виглядати ідентично у зашифрованому вигляді, що в свою чергу створює небезпеку відтворення вхідної частини інформації. Наприклад, при обробці тексту, який має високу інформаційну навантаженість, за допомогою алгоритму незалежного типу, у результуючому зашифрованому тексті з великою імовірністю можна буде спостерігати однакові частини, які повторюються. Аналізуючи характер таких зашифрованих даних, з високою імовірністю може бути відтворена вхідна інформація. За умови, що зловмисник має вхідний та вихідний блок даних, виконання відтворення ключа, який був використаний для криптографічного перетворення, а також злам алгоритму шифрування набагато спрощується. Алгоритми залежного типу характеризуються вищим ступенем захисту, проте поганою захищеністю від завад у каналах зв'язку, а також неможливістю обробляти частини інформаційного повідомлення паралельно. Іншими словами, швидкість реалізації синхронізованих криптографічних алгоритмів має суттєві обмеження, до того ж, характерна їм нестійкість до завад робить цей тип алгоритмів непридатним для шифрування інформаційних повідомлень, який передається в інформаційних каналах з високим рівнем шуму. Більша частина алгоритмів симетричного виду працюють в обох описаних вище режимах. Як приклад можна навести алгоритм DES, який виконує несинхронізоване опрацювання частин інформаційного повідомлення при режимі ECB, та синхронізоване опрацювання при режимах CFB, OFB та CBC. Алгоритми несиметричного виду виконують опрацювання частин лише несинхронізованим чином [23].

Ознаки алгоритмів симетричного виду, які часто використовуються для вирішення практичних завдань захисту інформації, такі як: довжина інформаційної частини, довжина ключа, технічні ресурси, які потрібно використати для зламу

алгоритму та отримання захищеної інформації, швидкість виконання програми алгоритму – зведені до таблиці 1.1.

Проаналізувавши та порівнявши симетричні алгоритми, можна дійти таких висновків:

1. Чітко простежується закономірність збільшення розрядності ключа в більш сучасних криптографічних алгоритмах, що є логічним, оскільки ключ більшої довжини може гарантувати вищий рівень стійкості алгоритму до зламу.

Таблиця 1.1

Ознаки алгоритмів симетричного виду

Алгоритм	Довжина інф. частини, байт	Довжина ключа, байт	Потрібно циклів	Потрібно спроб для зламу 2^k, k	Потрібно пам'яті для зламу 2^k, k	Продуктивність, Мбайт/с	Асемблер
DES	8	7	15	43	13	41	79
3-DES	8	14	46	56	56	16	30
IDEA	8	16	7	70	49	28	51
SAFER-64	8	8	7	46	32	46	98
Blowfish-16	8	10	15	32	32	84	117
RC5-128	16	16	18	–	–	51	104
SAFER-128	8	16	7	64	32	28	81
SHARK	16	16	4	–	–	30	42
SQUARE	16	16	6	73	32	88	137
Rijndael	16	32	12	–	–	19	38
RC-6	16	32	19	–	–	49	123

2. Сучасніші алгоритми мають кращу продуктивність, що пояснюється відсутністю використання в їх реалізації таких елементів, як складні лінійні перетворення та перестановки. Так, вони базуються на неякісних з криптографічного погляду перетвореннях. Але перевагою таких перетворень є те, що їх апаратна реалізація є ефективною. Як приклад можна навести виконання нелінійних перетворень застосуванням операцій цілочисельного додавання, віднімання та множення, та операцій циклічних зсувів для виконання перестановок у сучасному алгоритмі RC-6. Неоптимальні з криптографічного погляду перетворення призводять до того, що кількість ресурсів, витрачених на злам алгоритму, на практиці значно менша теоретично обрахованої межі, яка залежить від значення розрядності ключа. Так, наприклад, теоретично межею затрат обчислювальних ресурсів для зламу алгоритму IDEA при розрядності ключа 128 є 2^{128}, проте практично цей показник складає 2^{70}.
3. У реалізації більшості сучасних алгоритмів ключі формуються завчасно, за рахунок чого інформація не лише шифрується швидше, а й довжину ключів, які використовуються для шифрування, можна формувати та змінювати більш гнучко.
4. У сучасних криптографічних алгоритмах захисту даних, аби ускладнити диференціальний криптографічний аналіз, в методах прописуються варіації виконання, які залежать від кодів, отриманих в процесі здійснення перетворення.
5. Зростання використання блоків нелінійного перетворення з більшим розміром зменшило кількість циклів, необхідних для виконання перетворення.

Із алгоритмів незворотного виду, які застосовуються в комп’ютерних системах та мережах, можна виділити лише хеш-алгоритми. Робота цих алгоритмів

передбачає формування хеш-сигнатур-кодів з конкретною довжиною, яка диктується розрядами повідомлень або файлами довільного розміру. Можна виділити два класи хеш-алгоритмів: перший використовує шифроблоки алгоритмів симетричного виду, другий виконує рекурсивне обчислення систем булевих функцій.

За видом наявних зв'язків між шифроблоками, які є основою класу хеш-алгоритмів, ці алгоритми можна поділити на ті, які мають лінійну структуру, та ті, яким характерна перехресна структура [18]. Зазвичай у хеш-алгоритмах, які мають лінійну структуру, розрядність сформованої хеш-сигнатури дорівнює результуючій розрядності шифроблоку, який використовується. Прикладами хеш-алгоритмів, шифроблоки яких з'єднані лінійно, є алгоритми Девіса та Рабина [24]. Структура хеш-алгоритмів, шифроблоки в яких з'єднані перехресно, є складнішою. Як правило, в архітектурі таких алгоритмів шифроблоки формують дві чи чотири «лінії», які з'єднані перехресним способом, схожим на мережу Файстеля [22]. Тому розрядність результуючої хеш-сигнатури дорівнює вдвічі чи в чотири рази збільшеній розрядності шифроблоку. Прикладами хеш-алгоритмів з перехресною структурою є алгоритми MDC-2 та MDC-4.

Значення основних ознак для хеш-алгоритмів, які широко використовуються на практиці, представлено в таблиці 1.2.

Булеві функції використовуються для створення незворотних перетворень, які є основою симетричних алгоритмів, переважної кількості хеш-алгоритмів, а також засобів, які створюють бінарні псевдовипадкові послідовності. Принцип їх роботи такий: спочатку виконується опрацювання інформаційного блоку *data*, наступним кроком формується система булевих функцій: для хеш-алгоритмів – $\varphi_k(data)$, $k=1,...,m$, для алгоритмів шифрування – $\psi_k(data, key)$, де *key* – код ключа. Для того, щоб обчислити систему булевих функцій, залежну від великої кількості змінних, адже порядок розрядності двійкових векторів *data* та *key* дорівнює сотням,

необхідно використовувати рекурсію, при цьому обрахунок буде здійснено за декілька циклів.

Таблиця 1.2

Ознаки хеш-алгоритмів

Хеш-алгоритм	Довжина інф. частини, байт	Довжина хеш-сигнатури, байт	Продуктивність, Мбайт/с
PIREMD-160	64	20	5
SHA-1	64	20	6.125
MD4	64	16	20.875
MD5	64	16	14.25
MASH-1	16	16	0.55
MASH-2	16	16	0.025
MDC-2	8	16	1
MDC-4	8	16	0.5
SNEFRU	8	16	1.25
ГОСТ R-34.11-94	8	32	0.725

До структури кожного циклу входять такі елементи, як почергові операції нелінійних булевих перетворень, змішування бітів та виконання перемішування з блоком даних – для хеш-алгоритмів – чи ключем – для алгоритмів шифрування. Структуру обчислювальних процедур, основою яких є булеві функціональні перетворення, узагальнено на рис. 1.2.

Для нелінійних перетворень необхідна їх відповідність таким критеріям, як: строгий лавинний ефект [14] та велике значення нелінійності [11]. Як наслідок таким перетворенням характерна висока обчислювальна складність. Для алгоритмів симетричного виду нелінійні перетворення здійснюються над частинами вектору станів, та практично найбільш часто виконується у вигляді

таблиць. Програмна реалізація нелінійних перетворень, які використовуються у хеш-алгоритмах, здійснюється виконанням логічних операцій над словами, розрядність яких дорівнює 32. Таким чином, аби значення нелінійності та диференціальної ентропії були достатньо високими потрібно робити більшу кількість циклів, ніж при використанні симетричних алгоритмів. Для симетричних алгоритмів кількість циклів складає 10-18, в той самий час, як для хеш-алгоритмів це значення дорівнює 70-90. Для виконання змішування бітів виконуються поодиноко або у зв'язці такі операції, як: лінійні операції (Rijndael), зсуви (Rijndael, RC-6, SHA-1), пряма перестановка бітів (DES).

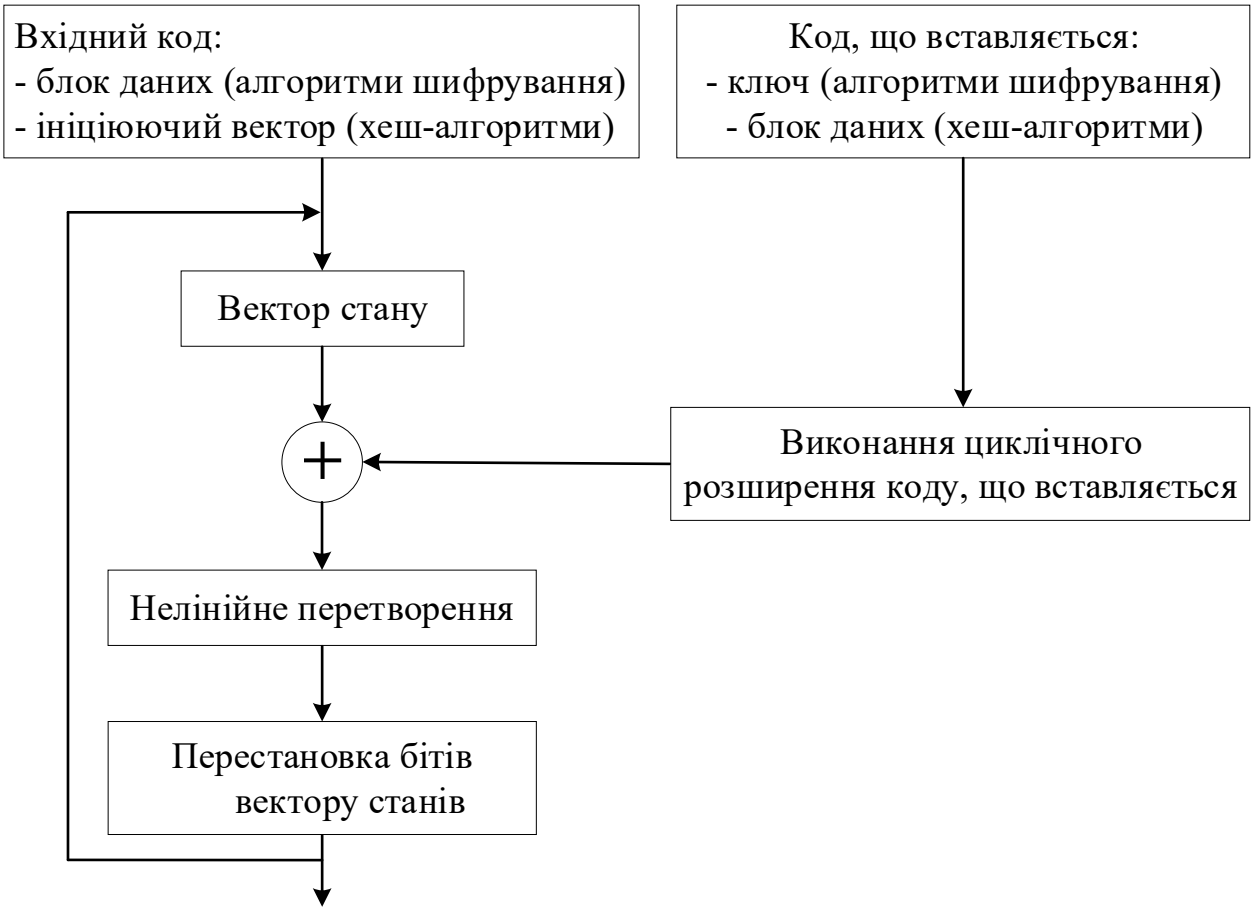

Рис. 1.2. Узагальнена структура обчислювальних процедур алгоритмів, основою яких є булеві функціональні перетворення

1.3. Нелінійність булевих перетворень. Огляд методів тестування криптографічних алгоритмів на стійкість до лінійного крипто аналізу

Тестування криптографічних алгоритмів є одним з важливих етапів їх розробки, оскільки дає можливість об’єктивно оцінити рівень захисту даних у комп’ютерних системах та мережах, який забезпечує конкретний алгоритм. Як правило, обчислення рівня захисту базується на визначенні кількості обчислювальних ресурсів, що були витрачені для зламу алгоритму. Процес тестування ускладнюється за рахунок варіативності сценаріїв зламу, які використовуються на практиці [42, 53]. Наприклад, відомою для зловмисника частиною при криптографічному аналізі можуть бути або лише результуючий текст, або один блок (чи декілька блоків) вхідного тексту та результуючий текст, розташування відомих блоків вхідного тексту може бути суміжне або ні, існування можливості ввести один блок (чи декілька блоків) вхідного тексту. До різних типів алгоритмів застосовують відмінні технології зламу. Так, аби зламати алгоритм симетричного типу необхідно визначити код ключа, для зламу хеш-алгоритму потрібно знайти таке повідомлення, яке призведе до колізії, в свою чергу, для порушення захисту, який надає алгоритм потокового типу, необхідно побудувати відтворюючу модель. Національні стандарти та стандарти ISO вимагають обов’язкове проходження тестування для алгоритмів захисту даних різного типу [26].

Розглянемо поширені методи, за допомогою яких аналізується стійкість криптографічних алгоритмів до зламів.

Найочевиднішим способом порушення захисту є спосіб простого перебору, який виконується таким чином: одна за одною виконується зміна кодів невідомого фрагменту, вхідним значенням слугує фрагмент шифру, який є вже відомим, далі виконується обчислення вихідного тексту за конкретним алгоритмом, останнім

кроком здійснюється порівняння отриманого вихідного тексту з вихідним текстом, який було задано. У випадку алгоритмів симетричного типу виконується перебір значень ключа, а для хеш-алгоритмів – вхідного блоку даних. На практиці підбирають лише один шифроблок. Як приклад розглянемо процес зламу алгоритму DES, який працює в несинхронізованому режимі ECB: за умови, що вхідний та вихідний тексти блоку є відомими, аби визначити невідомий елемент перебираються 2^{56} потенційних варіантів ключа, який має розрядність 56. Розпаралелити задачу зламу криптографічних алгоритмів шляхом перебору можна за допомогою використання універсальних комп'ютерів, які об'єднані в одну мережу, або спеціалізованих систем, які складаються з багатьох процесорів, причому розподіл наборів кодів ключів для перебору виконується таким чином, аби вони не пересікалися [12].

Застосування таких методів, як диференціальний та лінійний криптографічні аналізи може зменшити кількість варіантів ключа, які потрібно перебрати для виконання зламу криптографічного алгоритму. Ці загальні аналітичні методи дозволяють достатньо ефективно здійснювати процес порушення захисту даних.

Метод диференціального криптографічного аналізу був запропонований наприкінці IXX століття [17]. Його ідея базується на аналізі такого параметру, як імовірнісні оцінки, які показують наскільки зміни у вхідній інформації шифроблоку впливають на зміну вихідної інформації, що дає можливість отримати найімовірніший варіант коду закритого ключа, виконавши при цьому відносно невеликий перебір. Спочатку описаним методом здійснили злам DES, згодом він показав ефективність при виконанні зламів й інших алгоритмів.

З технологічної точки зору можна виділити два етапи, на які поділяється диференціальний криптографічний аналіз. Першим етапом є побудова таблиці імовірнісних оцінок того, наскільки зміна вхідних даних впливає на вихідні дані, другим етапом є підбір закритого ключа. Аби побудувати таблицю імовірнісних

оцінок необхідно сформувати множину Λ, до якої входять усі можливі зміни пар вхідної інформації шифроблоку, та множину Θ, до якої входять усі можливі зміни пар вихідної інформації шифроблоку. Аби сформувати множини Λ та Θ потрібно визначити максимально можливу відмінність між вхідними та вихідними змінами. Для прикладу, якщо вхідні дані шифроблоку мають розрядність l, а максимальна кількість відмінних розрядів – m, тоді множина Λ складається з C_l^m елементів. Так само можна визначити, скільки елементів входить до множини Θ. Коли були сформовані множини Λ та Θ, заповнюється таблиця відповідності вхідних та вихідних змін. Для заповнення цієї таблиці виконується формування k пар векторів вхідної інформації, які складають множину Ψ з однаковими різницями: $\underline{D} = (d_1,d_2,...,d_n) \in \Lambda$, тоді $<\underline{R}_1, \underline{R}_2> \in \Psi$: $r_{1h} \oplus r_{2h} = r_h$, $h = 1, ..., l$. На один вхід шифроблоку надходять вхідні вектори $\underline{R}$, а на інший – постійний код ключа, на виході шифроблоку отримується вихідний код $\underline{O}$. Кожній парі $<\underline{R}_1, \underline{R}_2> \in \Psi$ відповідає пара $<\underline{O}_1, \underline{O}_2>$: $o_{1h} \oplus o_{2h} = u_h$, $h = 1, ..., l$, $\underline{U} = (u_1, u_2, ..., u_n) \in \Theta$. Частотність відповідності $\delta(\underline{R}, \underline{O})$ дорівнює відношенню кількості ζ пар $<\underline{O}_1, \underline{O}_2>$, відмінність яких відображає вектор $\underline{U} \in \Theta$, до сумарного числа k пар векторів вхідної інформації, відмінність яких відображає вектор $\underline{D}$: $\delta(\underline{R}, \underline{O}) = \zeta/k$. Подібні дії виконуються над усіма векторами відмінностей, які входять до множини Λ.

Побудована таблиця та виконання тестових спроб використовуються для підбору найбільш задовільного ключа. Аналіз спроб здійснюється в залежності від алгоритму, злам якого виконується описаним способом.

Найвищу ефективність метод диференціального криптографічного аналізу демонструє при виконанні зламу булевих перетворень, які є взаємно неоднозначними та складаються з одного каскаду. Зауважимо, що велика кількість циклів Фейстелових перестановок значно знижує ефективність роботи методу, а коли кількість циклів стає більше вісімнадцяти, розглянутий метод криптографічного аналізу стає майже рівнозначним принципу простого перебору.

На показник ефективності суттєво впливає можливість задання вхідних текстових блоків під час проведення криптографічного аналізу, оскільки при відсутності такої можливості, за умови того, що можливо аналізувати тільки випадково перехоплені фрагменти тексту, ефективність цього методу значно зменшується. Для ілюстрації наведемо показники ефективності, які демонструє метод диференціального криптографічного аналізу при зламі DES, який описаний у роботі [50]: коли при складанні таблиці ймовірнісних відповідностей відмінностей існує можливість довільно обирати текстові вхідні блоки, кількість необхідних вхідних блоків буде приблизно складати 2^{45}; коли ж можливим тільки є перехоплення випадкових текстових фрагментів, для проведення криптографічного аналізу потрібно буде використати вже 2^{54} вхідних блоків. Після того, як таблиця імовірнісних відповідностей була побудована, процес добирання ключів додатково займе 2^{35} циклів обрахунку DES. Отже, за умови здійснення достатньо суттєвого припущення з приводу довжини текстового фрагменту, який перехоплюється, проведення диференціального криптографічного аналізу скорочує кількість спроб підбору ключа у 2^{17} разів порівняно з повним перебром.

Ще одним поширеним методом, який дозволяє скоротити обсяг перебору при здійсненні зламу криптографічних алгоритмів симетричного виду, є лінійний криптографічний аналіз, що використовує принцип підбору лінійної апроксимації нелінійних функціональних перетворень. Алгоритм цього методу передбачає заміну блоку нелінійного перетворення його наближеним лінійним представленням, формування якого здійснюється під час проведення аналізу послідовностей тексту. Метод побудови наближеної лінійної функції дає можливість прискорити знаходження апроксимованого вирішення задачі зламу, відштовхуючись від якого здійснюється визначення точного вирішення виконанням додаткового перебору. Використання методу лінійного криптографічного аналізу при виконанні зламу DES виявилося більш ефективним у

порівнянні з методом диференціального криптографічного аналізу. Для зламу DES методом знаходження лінійного наближення в середньому знадобилося 2^{42} спроби, що на декілька порядків менше кількості спроб, які необхідно виконати методом диференціального криптографічного аналізу. Додатково збільшити ефективність здійснення зламу методом визначення лінійної апроксимації можна використавши спеціалізовані апаратні засоби [29].

Основою описаних методів, за допомогою яких можна ефективно скоротити перебір можливих кодів ключа під час виконання зламу алгоритмів, є властиві булевим функціям, на основі яких будуються криптографічні перетворення, ознаки. Спираючись на цей факт, непрямим способом визначення рівня стійкості криптографічних алгоритмів до зламу диференційним та лінійним криптографічними аналізами є оцінка таких параметрів, які характеризують булеві функціональні перетворення, як нелінійність та диференціальні властивості булевих функцій, які є основою конкретного алгоритму. Досліджено, що використовуючи систему булевих функцій, яка є рівнозначною криптографічному перетворенню, що використовує алгоритм захисту даних, можна гарантувати стійкість криптографічного алгоритму до зламів диференційним та лінійним криптографічними аналізами, за умови незалежності функцій між собою та їх відповідності властивостям збалансованості та максимально можливої нелінійності, а також критерію SAC (Strict Avalanche Criterion – критерій строгого лавинного ефекту) [21, 32].

Булева функція $\beta(b_1,..., b_m)$ від m змінних вважається збалансованою (або такою, яка володіє максимальною ентропією) за умови, що значення цієї функції з ідентичною імовірністю приймають значення нуля та одиниці. Можна казати, що такій функції характерна максимальна ентропія, або що її таблиця істинності складається з однакової кількості нульових та одиничних значень: $\sum_{\forall \bar{B} \in W} \beta(\bar{B}) = 2^{m-1}$, де W є множиною усіх потенційних наборів вхідних змінних.

Булева функція відповідає критерію SAC (що свідчить про її максимальну диференціальну ентропію) за умови, що значення функції змінюється з імовірністю ½ при зміні значення кожної з вхідних змінних, тобто:

$$\sum_{\forall \bar{B} \in W} \left(\beta(b_1, \dots, b_k, \dots, b_m) \oplus \beta\left(b_1, \dots, \overline{b_k}, \dots, b_m\right)\right) = 2^{m-1}, \forall k \in \{1, \dots, m\}$$

Продуктивність методу диференціального криптографічного аналізу зростає за умови, що булева функція, яка є рівнозначною криптографічному перетворенню алгоритму захисту інформації, не відповідає критерію SAC. Одним з кроків процесу диференціального криптографічного аналізу може бути побудова таблиці відмінних змін між двома чи трьома функціями. Аби протидіяти цьому виду диференціального криптографічного аналізу булеві функції повинні задовольняти критерій PC(n) (Propagation Criterion – критерій розповсюдження степені n), що показує їхню максимальну умовну диференціальну ентропію при зміні довільних n змінних:

$$\sum_{\forall \bar{B} \in W} (\beta(\bar{B}) \oplus \beta(\bar{B} \oplus \bar{Y})) = 2^{m-1}, \bar{Y} = \{y_1, \dots, y_m\}, y \in \{0,1\}, \sum_{j=1,n} y_j = n$$

За умови відповідності булевої функції від m змінних критерію розповсюдження PC(m) її називають bent-функцією. Така функція характеризується максимальним рівнем умовної диференціальної ентропії від довільної підмножини власних змінних, проте така функція не є збалансованою, а тому не може напряму бути використана для формування криптографічних перетворень.

Чим вищий показник нелінійності у криптографічних перетворень, тим більшу стійкість вони мають до зламів лінійним криптографічним аналізом. Булева функція називається лінійною, якщо в її алгебраїчній нормальній формі немає добутку змінних. Відстань Геммінга $HmD(\beta, \gamma)$ дорівнює кількості вхідних наборів $b_1, b_2, \dots b_m$ вектору B, які мають різні значення для функції $\beta(B)$ та лінійної функції $\gamma(B)$. Обчислення відстані Геммінга $HmD(\beta, \gamma)$ виконується за такою формулою:

$$HmD(\beta,\gamma) = \sum_{B\in Z}\big(\beta(B) \oplus \gamma(B)\big), \qquad (1.1)$$

де Z – множина потенційних векторів B, кількість яких дорівнює 2^m.

Значення нелінійності $NL(\beta)$ булевої функції $\beta(b_1,b_2,\ldots b_m)$ від m змінних визначається відстанню Геммінга між цією функцією та найближчою лінійною функцією, тобто мінімальною кількістю наборів, на яких булева функція $\beta(b_1,b_2,\ldots b_m)$ має інше значення в порівнянні з довільною з 2^{m-1} лінійних функцій, що можна обрахувати за такою формулою:

$$NL(\beta) = \min HmD(\beta,\gamma)$$

Булева функція, нелінійність якої дорівнює максимальному значенню, задовольняє критерій розповсюдження PC(n) та є bent-функцією.

Під час виконання тестування криптографічних алгоритмів перевіряється наскільки часткові булеві функції, які використовуються для формування нелінійних перетворень, відповідають описаним вище критеріям. Прикладами нелінійних перетворень є S-блоки для алгоритму DES та операції мультиплікативної інверсії на полях Галуа – для Rijndael. Здійснення такого тестування не є складним, адже побудова таблиць істинності для часткових функцій, які залежать не більше, ніж від восьми змінних, є простою задачею. Проте не можна стверджувати, що якщо булеві часткові функціональні перетворення мають специфічні ознаки, то і їх матимуть системи булевих функцій, які відповідають усьому криптографічному алгоритму. У свою чергу перевірити, чи такі системи функцій мають специфічні ознаки неможливо, бо кількість змінних, на яких вони визначені, може досягати порядку сотень. Через це на практиці визначення показників нелінійності та диференціальної ентропії алгоритмів захисту виконується статистичними методами. Чим більша множина значень функції, тим більша стабільність цих методів. На практиці кількість значень функції складає 10^3-10^5. Отже, для реалізації процедур лінійного та диференціального криптографічних аналізів для оцінки стійкості криптографічних алгоритмів

потрібно витратити досить суттєвий обсяг обчислювальних ресурсів порівняно зі зламом алгоритму. До того ж задача криптографічного статистичного дослідження критеріїв булевих перетворень є такою, яка добре розпаралелюється [20].

Беручи до уваги те, що етап тестування алгоритмів захисту даних за допустимий час є важливим при підтримці існуючих та розробці нових криптографічних алгоритмів, на сьогоднішній день було розроблено ряд методів [35, 48], які дозволяють пришвидшити процес обрахунку нелінійності булевих функціональних перетворень. Але практично всі методи пришвидшення знаходження нелінійності не виконують поставлену задачу повністю: вони визначають цей показник із завчасно вказаною похибкою, або знаходять тільки нижню границю нелінійності булевої функції, або тільки з'ясовують, чи є вона нелінійною.

Перший підхід використовує метод статистичної оцінки властивості нелінійності булевої функції, здійснюючи підбір наближеної до неї лінійної функції [7]. Алгоритм знаходження нелінійності за зазначеним методом складається з таких етапів:

- формування підмножини Θ лінійних функцій за визначеними правилами;
- знаходження в отриманій підмножині Θ лінійної функції, відстань між якою та заданою функцією є мінімальною.

Наперед вказана похибка ξ визначає розмір підмножини Θ. Крім того, з огляду на статистичний характер методу, існує ймовірність, що реальна похибка буде більше заданої.

Представником другого підходу є метод статистичного знаходження нижньої границі показника нелінійності булевої функції [8]. Цей метод вирішує задачу визначення нелінійності способом розділення множини W всіх лінійних функцій, кількість яких дорівнює 2^{m+1}, на підмножину аналогічних до вхідної булевої функції та підмножину відмінних від неї лінійних функцій на μ наборах вхідних змінних.

Від значення параметру μ залежить наскільки точно буде визначено нижню границю нелінійності, що допускає варіант здійснення практично повного перебору.

Виконавши оглядовий аналіз розроблених методів прискореного тестування криптографічних алгоритмів на основі булевих перетворень, можна зробити висновок, що розглянуті існуючі на сьогодні методи прискореного визначення нелінійності булевих перетворень не повністю задовольняють задачу тестування сучасних криптографічних алгоритмів.

Висновки до розділу 1

У першому розділі магістерської дисертації були проведені огляд та аналіз методів оцінки криптографічної захищеності алгоритмів криптографічного захисту даних, а також основні напрямки розвитку щодо підвищення цього захисту, на основі яких були сформульовані такі висновки:

1. Рівень криптографічної стійкості алгоритмів захисту даних до зламів є комплексним показником: покращення одної складової може призвести до погіршення інших і, як результат, рівня криптографічної стійкості в цілому. Зважаючи на це, вибір складових показнику криптографічної стійкості повинен базуватися на особливостях використання конкретного алгоритму.

2. Було окласифіковано та охарактеризовано основні види алгоритмічних криптографічних механізмів захисту даних. Було виявлено, що використання ключа з більшою бітовою довжиною призводить до підвищення криптографічного рівня захисту алгоритмів симетричного виду та їх стійкості до зламів. З іншого боку, виключення елементів перестановки та використання замість складного нелінійного перетворення більш простого, дозволяє отримати виграш у швидкодії роботи алгоритмів симетричного виду. При цьому було показано, що для переважної частини алгоритмів, основою яких є нелінійні булеві перетворення, головними є такі критерії, як нелінійність та SAC.

3. Аналіз наявних методів для пришвидшення обрахунку показнику нелінійності булевих перетворень виявив, що вони не в змозі надати належний рівень тестування через те, що параметри для роботи цих методів задаються безпосередньо перед початком їхнього виконання. Таким чином, для підвищення ефективності тестування сучасних криптографічних алгоритмів нагальною є потреба розробки новітніх більш точних методів обрахунку показнику нелінійності.

РОЗДІЛ 2

РОЗРОБКА МЕТОДУ ПРИСКОРЕНОГО ТЕСТУВАННЯ НЕЛІНІЙНОСТІ КРИПТОГРАФІЧНИХ ПЕРЕТВОРЕНЬ

2.1. Теоретичне обґрунтування методу

Для вирішення задачі прискорення тестування показнику нелінійності булевих перетворень запропонований метод виконує динамічне відновлення найбільш наближеної лінійної функції $\gamma_0(B) = \alpha_1 \cdot b_1 + \alpha_2 \cdot b_2 + \cdots + \alpha_m \cdot b_m$ до вхідної збалансованої булевої функції $\beta(b_1, b_2, \dots, b_m)$, де B – набір булевих змінних, кількість яких дорівнює m: $B=\{b_1,b_2,\dots b_m\}$, $\forall k \in \{1,2,\dots m\}: b_k \in \{0,1\}$, Визначення такої лінійної функції здійснюється за допомогою обчислення відстані Геммінга між кожною з множини лінійних функцій та вхідною нелінійною функцією. Тобто результатом роботи методу є значення коефіцієнтів $\alpha_1,\alpha_2,\dots,\alpha_m$ найближчої лінійної функції $\gamma(B)$.

Ідея методу полягає в послідовному знаходженні змінних $b_1,b_2,\dots,b_m$, з яких складається АНФ лінійної функції $\gamma_0(B)$, значення відстані Геммінга між якою та вхідною нелінійною збалансованою функціє $\beta(b_1, b_2, \dots, b_m)$ є найменшим.

Вибір змінних, які є доданками в АНФ лінійної функції $\gamma_0(B)$, здійснюється таким способом: обраховуються імовірності r_l того, чи зміниться значення булевої функції, якщо інвертувати l-ну змінну b_l вхідної булевої функції $\beta(b_1, b_2, \dots, b_m)$. Обчислення цього критерію виконується за такою формулою:

$$r_l = {}^{1}/_{2^m} \cdot \sum_{B \in Z} \beta(B) \oplus \beta(B \oplus U_i), \qquad (2.1)$$

де Z – множина потенційних векторів B, кількість яких дорівнює 2^m, $U_l=\{u_{l1},u_{l2},\dots,u_{lm}\}$ – m-розрядний бінарний вектор, елемент l якого має одиничне значення, усі інші – нульове: $\forall s \in \{1,..,l\text{-}1,l\text{+}1,\dots,m\}: u_{ls} = 0, u_{ll} = 1$.

Логічно, що завжди зміна значення компоненти b_l, яка входить лінійно до складу АНФ булевої функції $\beta(b_1, b_2, \dots, b_m)$, що можна виразити таким чином: $\beta(B) = b_l \oplus \rho(b_1, \dots b_{l-1}, b_{l+1}, \dots, b_m)$, де $\rho(b_1, \dots b_{l-1}, b_{l+1}, \dots, b_m)$ – булева функція, незалежна від змінної b_l, буде призводити до зміни значення вхідної булевої функції $\beta(b_1, b_2, \dots, b_m)$, інакше кажучи $r_l = 1$.

Отже, значення імовірності r_l того, що значення вхідної функції $\beta(b_1, b_2, \dots, b_m)$ зміниться за умови інвертування значення змінної b_l практично є імовірністю факту входження змінної b_l до складу АНФ лінійної функції $\gamma_0(b_1, b_2, \dots, b_m)$, значення відстань Геммінга між якою та функціє $\beta(b_1, b_2, \dots, b_m)$ є найменшою. Тобто, значення вектору R=$\{r_1,r_2,\dots r_m\}$ вказує імовірність того, що двійкові коефіцієнти $\alpha_1,\alpha_2,\dots,\alpha_m$ лінійної функції $\gamma(B)$ будуть мати одиничне значення.

Використавши значення вектору R, можна пришвидшити реконструювання лінійної функції, виконавши спрямований перебір значень бінарних коефіцієнтів $\alpha_1,\alpha_2,\dots,\alpha_m$ таким чином: початкові значення коефіцієнтів обираються згідно значень елементів вектору R. Тобто: $\alpha_l = 1$ якщо $r_l > 0{,}5$ та $\alpha_l = 1$ якщо $r_l \leq 0{,}5$, $\forall l \in \{1,2,\dots,m\}$.

Наступним кроком за формулою (1.1) здійснюється обчислення відстані Геммінга $HmD(\beta, \gamma)$ між вхідною функцією $\beta(B)$ та лінійною функцією $\gamma(B) = \alpha_0 + \alpha_1 \cdot b_1 + \alpha_2 \cdot b_2 + \dots + \alpha_m \cdot b_m$ з фактичними коефіцієнтами $\alpha_1,\alpha_2,\dots,\alpha_m$.

Лінійна функція $\gamma(B)$ вважається найближчою до вхідної нелінійної функції $\beta(B)$, тобто відстань Геммінга між функціями $\gamma(B)$ та $\beta(B)$ є мінімальною, при виконанні такої умови:

$$HmD(\beta, \gamma) = 2^{m-2} \qquad (2.2)$$

Якщо умова (2.2) не виконується, то обраховане значення відстані Геммінга $HmD(\beta, \gamma)$ встановлюється як мінімальне $HmD_{min} = HmD(\beta, \gamma)$, а коефіцієнти $\alpha_1,\alpha_2,\dots,\alpha_m$ лінійної функції змінюються таким чином: проводиться аналіз значень

елементів вектору імовірностей R, якщо значення елементу r_q наближене до 0,5, то виконується послідовне інвертування значення коефіцієнту α_q, $\forall q \in \{1,2,\ldots,m\}$. Після чого виконується обчислення відстані Геммінга $HmD(\beta, \gamma)$ між вхідною функцією $\beta(B)$ та оновленою лінійною функцією $\gamma(B)$. За умови, що обрахована відстань Геммінга, менше поточного мінімального значення: $HmD(\beta, \gamma) \leq HmD_{min}$ – змінене значення обраного коефіцієнту зберігається і виконується перевірка умови (2.2). Якщо обраховане значення $HmD(\beta, \gamma)$ відповідає цій умові, то пошук γ_0 закінчується. Якщо ні, то продовжується підбір коефіцієнтів $\alpha_1,\alpha_2,\ldots,\alpha_m$ зазначеним вище способом.

Експериментально досліджено, що процес обрахунку показнику нелінійності за допомогою поступового підбирання лінійної функції $\gamma_0(B)$, відстань Геммінга між якою та функцією $\beta(B)$ є мінімальною, описаним методом є ефективнішим за умови, коли значення елементів $r_1,r_2,\ldots r_m$ вектору R є значно різними.

Організувати підбір лінійної булевої функції $\gamma_0(B)$ з мінімальною відстанню Геммінга до нелінійної булевої функції $\beta(B)$ шляхом аналізу елементів вектору R імовірностей того, що інвертування змінних функції $\beta(B)$ призведе до зміни її значення, ефективніше за допомогою концепцій динамічного програмування.

За цим методом виконується поетапне формування наближених до вхідної функції $\beta(B)$ лінійних функцій, залежних від 1-ої, 2-ох, 3-ох, тощо змінних, для відновлення лінійної функції $\gamma(B)$. Разом з цим, порядок вибору змінних, які будуть входити до фінального варіанту лінійної функції $\gamma(B)$, залежить від значень відповідних імовірностей зміни значення вхідної функції $\beta(B)$ в наслідок інвертування її змінних. Тобто, на початку для роботи обирається така лінійна функція $\gamma_1(b_t) = b_t$, до складу якої входить лише змінна b_t з найбільшим значенням елементу r_t вектору $R=\{r_1,r_2,\ldots r_m\}$. Після чого виконується формування множини Ω_2 лінійних функцій, до складу яких входять дві змінні $\gamma_2(b_p, b_u)$, які мають максимальні значення елементів r_p та r_u відповідно: $r_u \leq r_p$, $r_u = \max\{r_1, r_2,\ldots, r_{p-1},$

$r_{p+1},\ldots, r_m\}$. Водночас до множини Ω_2 входять тільки ті наближені лінійні функції $\gamma_2(b_p, b_u)$, які є ефективними для вхідної нелінійної функції $\beta(B)$. Описаним способом виконується перебір варіантів лінійних комбінацій функцій $\gamma_2(b_p, b_u)$ з множини Ω_2 зі змінною b_g, яка має максимальне після r_p та r_u значення r_g вектору R, та отримується множина Ω_3, яка складається з наближених до вхідної функції $\beta(B)$ лінійних функцій, залежних від трьох змінних. При цьому відбір лінійних комбінацій $\gamma_2(b_p, b_u)\in\Omega_2$ з b_g до множини Ω_3 здійснюється на основі відстані Геммінга до вхідної функції $\beta(B)$, вона має бути мінімальною.

Робота алгоритму продовжується m кроків, поки не будуть опрацьовані усі змінні $b_1,b_2,\ldots,b_m$, відповідно до їх значень імовірностей $r_1,r_2,\ldots r_m$ у порядку спадання. Як наслідок роботи методу буде сформовано множину $\Omega_m(b_1,b_2,\ldots,b_m)$, до складу якої входять усі можливі лінійні функції від змінних $b_1,b_2,\ldots,b_m$, відстань Геммінга між якими та вхідною функцією $\beta(B)$ є мінімальною.

Цей підхід є ефективним, оскільки на кожному кроці зі збільшенням значення l темп зростання розміру множини Ω_l є повільнішим, тобто розмір цієї множини збільшується не вдвічі, а в рази повільніше.

2.2. Розробка базових процедур методу

Алгоритм роботи запропонованого методу складається з таких кроків:

1. Для кожної змінної виконується обчислення відповідних значень елементів вектору імовірностей $R=\{r_1,r_2,\ldots r_m\}$ за формулою (2.1).
2. Значення $r_1,r_2,\ldots r_m$ вектору R сортуються в порядку зменшення. Відсортовані значення $r_1,r_2,\ldots r_m$ позначаються як $\omega(i_1),\omega(i_2),\ldots,\omega(i_m)$, $\omega(i_1)\geq\omega(i_2)\geq\ldots\geq\omega(i_m)$, де i_1 – порядковий номер змінної вектору R, значення імовірності r_{i_1} якої найбільше, i_2 – порядковий номер змінної вектору R, значення імовірності якої друге за величиною після r_{i_1}, i_m – порядковий номер змінної вектору R, значення імовірності якої є найменшим.

3. Формується множина Ω лінійних функцій, до якої входить один елемент: $\gamma_1 = b_{i_1}$. Поточне значення кількості q функцій у множині Ω рівне одиниці: $q = 1$.
4. Індекс v перебору змінних імовірностей $\omega(i_1),\omega(i_2),\ldots,\omega(i_m)$, відсортованих по спаданню, встановлюється у значення 2: $v = 2$.
5. Індексу w перебору лінійних функцій, які входять до множини Ω, призначається значення 1: $w = 1$. Поточне значення кількості c складових, доданих до множини Ω на v-тому кроці, рівне нулю: $c = 0$.
6. Виконується обрахунок відстаней Геммінга $HmD(\gamma_w)$ та $HmD(\gamma_w \oplus b_{i_v})$.
7. Якщо виконується нерівність $HmD(\gamma_w \oplus b_{i_v}) \leq HmD(\gamma_w)$, тоді $\gamma_w = \gamma_w \oplus b_{i_v}$, інакше якщо лінійна функція $\gamma = b_{i_v}$ не входить до множини Ω: $b_{i_v} \notin \Omega$, тоді виконується додавання лінійної функції $\gamma_{q+1} = b_{i_v}$ до множини Ω. Інкрементуються значення кількості q функцій у множині Ω та кількість c елементів, доданих до множини Ω на v-тому кроці: $q = q + 1$, $c = c + 1$.
8. Обробка наступної лінійної функції з множини Ω шляхом збільшення відповідного індексу w на 1: $w = w + 1$. Якщо виконується нерівність $w \leq (q - c)$, тоді перехід на пункт 6.
9. Обробка наступної змінної з відсортованої за спаданням множини $\omega(i_1),\omega(i_2),\ldots,\omega(i_m)$ шляхом збільшення відповідного індексу v на 1: $v = v + 1$. Якщо виконується нерівність $v \leq m$, тоді перехід на пункт 5.
10. Виконання обрахунку значень відстаней Геммінга $HmD_1,HmD_2,\ldots,HmD_m$ для лінійних функцій $\gamma_1,\gamma_2,\ldots,\gamma_m$ з множини Ω. Визначення найменшого серед обрахованих значень.

Для прикладу візьмемо таку вхідну збалансовану нелінійну функцію $\beta(B) = b_4 \oplus b_3 \oplus b_3b_4 \oplus b_3b_4b_5 \oplus b_2b_4b_5 \oplus b_2b_3b_4 \oplus b_1 \oplus b_1b_5 \oplus b_1b_4 \oplus b_1b_4b_5 \oplus b_1b_3 \oplus b_1b_3b_5 \oplus b_1b_2 \cdot b_5 \oplus b_1b_2b_4 \oplus b_1b_2b_3 \oplus b_1b_2b_3b_5 \oplus b_1b_2b_3b_4 \oplus 1$, яка визначена на наборі булевих

змінних $B=\{b_1,b_2,\dots b_5\}$, $m = 5$. У форматі таблиці істинності ця функція має такий вигляд: 1100 0001 1101 0011 0100 0011 0011 1110. За допомогою формули (2.1) для кожної змінної набору B обчислюємо значення імовірності того, що інвертування цієї змінної призведе до зміни значення функції $\beta(B)$: $r_1=0.25$, $r_2=0.75$, $r_3=0.75$, $r_4=0.5$, $r_5=0.5$.

Виконується сортування в порядку спадання обрахованих значень імовірностей відповідно другого пункту алгоритму: $\omega(i_1)=0.75$, $\omega(i_2)=0.75$, $\omega(i_3)=0.5$, $\omega(i_4)=0.5$, $\omega(i_5)=0.25$. Відповідно змінні впорядкованої множини мають такі індекси значень імовірностей: $i_1=2$, $i_2=3$, $i_3=4$, $i_4=5$, $i_5=1$.

Далі, у відповідності з третім пунктом алгоритму, до множини Ω перспективних лінійних функцій додається лінійна функція, до складу якої входить єдина змінна з найбільшим значенням імовірності: $\gamma_1 = b_{i_v} = b_2$, значення поточної кількості q змінних в множині Ω дорівнює 1: $q = 1$. Індекс v перебору змінних імовірностей, відсортованих по спаданню, встановлюється у значення 2: $v = 2$. Індексу w перебору лінійних функцій, які входять до множини Ω, присвоюється значення 1: $w = 1$, а значення кількості c елементів, доданих до множини Ω на поточному кроці, встановлюється в 0: $c = 0$.

За пунктом 6 алгоритму виконується обчислення значень відстаней Геммінга $HmD(\gamma_w)$ та $HmD(\gamma_w \oplus b_{i_v})$:

$$HmD(\gamma_w) = HmD(\gamma_1) = HmD(b_2) = 12$$

$$HmD\left(\gamma_w \oplus b_{i_v}\right) = HmD\left(\gamma_1 \oplus b_{i_2}\right) = \left[b_{i_2} = b_3\right] = HmD(b_2 \oplus b_3) = 16$$

Нерівність $HmD(b_2 \oplus b_3) \leq HmD(b_2)$ не виконується та лінійна функція b_3 ще не входить до множини Ω: $b_3 \notin \Omega$, тому згідно сьомого пункту алгоритму до множини Ω додається ця лінійна функція: $\gamma_2 = b_3$, кожне зі значень кількості q елементів множини Ω та кількості c елементів, доданих до цієї множини на поточному кроці, збільшується на 1: $q = 2$, $c = 1$.

За восьмим пунктом алгоритму виконується перехід до наступної лінійної функції множини Ω шляхом інкрементування індексу w: w = 2 – та перевірка нерівності $w \leq (q - c)$, яка не виконується. Далі переходимо до наступного елементу множини імовірностей, впорядкованих за спаданням, шляхом збільшення значення індексу v на 1: v = 3, а також переконавшись, що умова $v \leq m$ в дев'ятому пункті не виконується.

Згідно з п'ятим пунктом встановлюють значення індексу w та кількості c лінійних функцій: w = 1, c = 0. Виконується обчислення відстаней Геммінга для функцій $\gamma_w = \gamma_1 = b_2$ та $\gamma_w \oplus b_{i_v} = \gamma_1 \oplus b_{i_3} = b_2 \oplus b_4$:

$HmD(b_2) = 12$

$HmD(b_2 \oplus b_4) = 14$

До множини Ω додається лінійна функція: $\gamma_3 = b_4$, оскільки нерівність (п. 7) не виконується та функція $b_4 \notin \Omega$, кожне зі значень кількості q елементів множини Ω та кількості c елементів, доданих до цієї множини на поточному кроці, збільшується на 1: q = 3, c = 1. Після збільшення індексу w на одиницю: w = 2 – нерівність (п. 8) виконується, тому обраховуємо:

$HmD(\gamma_w) = HmD(\gamma_2) = HmD(b_3) = 16$

$HmD(\gamma_w \oplus b_{i_v}) = HmD(\gamma_2 \oplus b_{i_3}) = [b_{i_3} = b_4] = HmD(b_3 \oplus b_4) = 10$

Нерівність (п. 7) є правдивою, тому лінійній функції γ_2 присвоюється нове значення: $\gamma_2 = b_3 \oplus b_4$. Значення w інкрементується: w = 3, нерівність (п. 8) не виконується, збільшуємо значення індексу v на 1: v = 4 – та перевіряємо нерівність (п.9). Ця нерівність виконується, тому переходимо до повторного виконання пункту 5: w = 1, c = 0.

Обчислені значення відстаней Геммінга для функцій $\gamma_w = \gamma_1 = b_2$ та $\gamma_w \oplus b_{i_v} = \gamma_1 \oplus b_{i_4} = b_2 \oplus b_5$ відповідно дорівнюють 12 та 14. Нерівність (п. 7) не виконується та функція $b_5 \notin \Omega$, тому до множини Ω додається лінійна функція:

$\gamma_4 = b_5$, значення q та c збільшуються на 1: q = 4, c = 1. Збільшивши індекс w на одиницю: w = 2, – та перевіривши, що нерівність (п. 8) виконується, обраховуємо:

$$HmD(\gamma_w) = HmD(\gamma_2) = HmD(b_3 \oplus b_4) = 10$$

$$HmD\left(\gamma_w \oplus b_{i_v}\right) = HmD\left(\gamma_2 \oplus b_{i_3}\right) = \left[b_{i_4} = b_5\right] = HmD(b_3 \oplus b_4 \oplus b_5) = 16$$

Нерівність (п. 7) не виконується та функція $b_5 \in \Omega$, значення q та c не змінюються: q = 4, c = 1. Збільшивши індекс w на одиницю: w = 3, – та перевіривши, що нерівність (п. 8) виконується, обраховуємо відстані Геммінга для функцій $b_3 \oplus b_4$ та $b_4 \oplus b_5$, отримаємо значення 14 та 16 відповідно.

Нерівність (п. 7) не виконується та функція $b_5 \in \Omega$, значення q та c залишаються попередніми: q = 4, c = 1. Збільшуємо індекс w на одиницю: w = 4, – перевіряємо, що нерівність (п. 8) не виконується, тому інкрементуємо значення індексу v на 1: v = 5 – та перевіряємо нерівність (п.9). Ця нерівність виконується, тому переходимо до повторного виконання пункту 5: w = 1, c = 0.

Обраховуємо відстані Геммінга $HmD(b_2) = 12$ та $HmD(b_2 \oplus b_1) = 16$. Нерівність (п. 7) не виконується та $b_0 \notin \Omega$, тому: $\gamma_5 = b_1$, до значень q та c додаємо 1: q = 5, c = 1. Збільшивши індекс w на одиницю: w = 2, – та перевіривши, що нерівність (п. 8) виконується, обраховуємо відстані Геммінга для функцій $b_3 \oplus b_4$ та $b_3 \oplus b_4 \oplus b_1$, отримаємо значення 10 та 10 відповідно.

Нерівність (п. 7) виконується, тому лінійній функції γ_2 присвоюється нове значення: $\gamma_2 = b_3 \oplus b_4 \oplus b_1$, – значення q та c залишаються попередніми: q = 5, c = 1. Збільшуємо індекс w на одиницю: w = 3, – та перевіряємо, що нерівність (п. 8) виконується, тому обраховуємо відстані Геммінга $HmD(b_4) = 14$ та $HmD(b_4 \oplus b_1) = 14$. Нерівність (п. 7) не виконується та функція $b_1 \in \Omega$, значення q та c залишаються попередніми: q = 4, c = 1.

Збільшуємо індекс w на одиницю: w = 4, – та перевіряємо, що нерівність (п. 8) виконується, тому обраховуємо відстані Геммінга для функцій b_5 та $b_5 \oplus b_4$, отримаємо значення 14 та 14 відповідно. Перевіряємо, що нерівність (п. 7) не

виконується та функція $b_1 \in \Omega$, значення q та c залишаються попередніми: $q = 4$, $c = 1$.

Збільшивши індекс w на одиницю: $w = 5$, – та перевіривши, що нерівність (п.8) не виконується, переходимо до наступного кроку алгоритму: інкрементуємо значення v: $v = 6$, – та перевіряємо, чи виконується нерівність (п.9). Ця нерівність не виконується, що свідчить про завершення формування множини Ω лінійних функцій.

Для обрахунку показнику нелінійності: nonlinearity = 10 – вхідної нелінійної булевої функції програма, використовуючи розроблений алгоритм, виконала 20 операцій порівняння та сформувала множини Ω наближених лінійних булевих функцій: $\Omega = \{b_2, b_3 \oplus b_4 \oplus b_1, b_4 \oplus b_1, b_5 \oplus b_1, b_1\}$.

2.3. Визначення характеристик ефективності методу та їх порівняльний аналіз з відомими методами тестування нелінійності

Для оцінки ефективності запропонованого методу прискорення визначення нелінійності булевих перетворень криптографічних алгоритмів захисту даних використовуються такі критерії [10], як:

- коефіцієнт прискорення χ обчислення показнику нелінійності, який отримується обчисленням такого співвідношення: кількість операцій порівняння, виконаних при використанні прямого перебору, на кількість операцій порівняння, виконаних при використанні розробленого методу;
- об'єм пам’яті, необхідний для програмної реалізації роботи методу;
- наскільки точним є обраховане значення нелінійності.

Для обчислення значення складності, тобто показнику того, як кількість виконаних операцій перебору залежить від кількості змінних m булевої функції, застосовують дані, отримані в результаті проведення експериментального статистичного моделювання розробленого методу тестування, оскільки складність

не можна виразити у формульному вигляді, бо обрахунок показнику нелінійності має непередбачуваний характер.

Проаналізувавши отримані експериментальні дані, було визначено, що розроблений метод має обчислювальну складність O(m^2).

З огляду на вищесказане, використання розробленого методу дає можливість зменшити об'єм обчислень у $2^m/_{m^2}$ разів порівняно з повним перебором. У якості прикладу можна навести порівняння значень кількості операцій перебору, виконаних під час проведення статистичного експериментального дослідження, з кількістю операцій, необхідних для виконання повного перебору, коли кількість аргументів булевої функції дорівнює 12: m = 12. Так, для повного перебору необхідно виконати 2^{12}=4096 операцій перебору, в той же час, як розробленому методу знадобилося всього 80 операцій, що дозволило зменшити значення обчислювальної складності в 51 раз. Коефіцієнти прискорення χ обчислення показнику нелінійності за допомогою розробленого методу відносно виконання повного перебору для функцій, залежних від різної кількості змінних m, зведено у таблиці 2.1.

Відмітимо, що пришвидшення визначення показнику нелінійності виконується за рахунок допущення наявності похибки, величина якої залежить від кількості змінних m. Характер зміни величини похибки в залежності від кількості змінних є неоднозначним, тому для її обрахунку було виконане статистичне дослідження.

Таблиця 2.1

Коефіцієнти прискорення визначення нелінійності для функцій від різної кількості змінних

Кількість змінних m	Кількість виконаних операцій перебору	Коефіцієнт прискорення χ
6	25	2,56
8	42	6,1
10	61	16,77
12	80	51,2
14	114	143,72
16	143	458,29
18	189	1 387,01

Значення величин похибки, отримані в результаті проведення статистичних експериментальних досліджень булевих функцій від різної кількості змінних m, показані на графіку на рис. 2.1. Проаналізувавши отриманий графік, можна зробити висновок, що середнє значення величини похибки обчислення нелінійності запропонованим методом стрімко спадає зі збільшенням кількості змінних. Наприклад, для функцій від 12 і більше змінних, які використовуються сучасними криптографічними алгоритмами, значення похибки незначне і становить у середньому приблизно 1%.

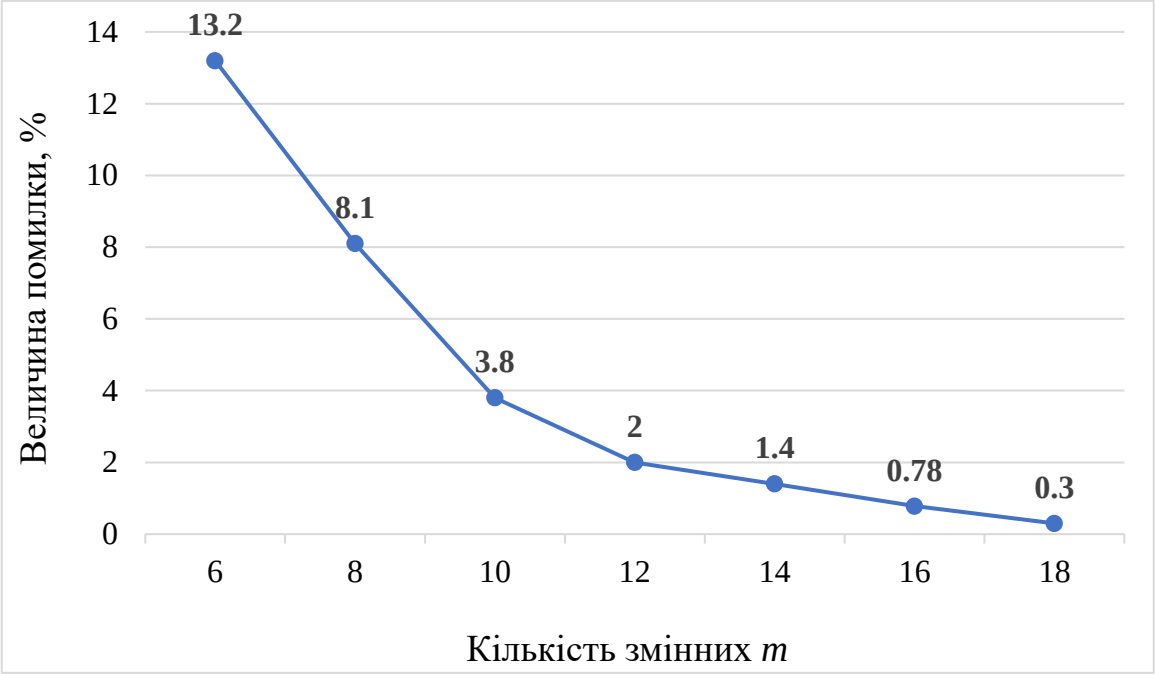

Рис. 2.1. Залежність значення помилки від кількості змінних m

Висновки до розділу 2

Другий розділ магістерської дисертації присвячений опису та оцінці ефективності розробленого методу тестування криптографічної стійкості алгоритмів захисту даних, основою яких є нелінійні булеві перетворення. Запропонований метод вирішує описану задачу за допомогою пришвидшення обрахунку показнику нелінійності булевих перетворень.

У цьому розділі були поставлені та виконані такі завдання:

1. Були описані теоретичні засади розробленого методу, який виконує послідовне відновлення найбільш наближеної лінійної функції, використовуючи концепції динамічного програмування.

2. Результати експериментального статистичного моделювання показали, що розроблений метод виконує набагато меншу кількість операцій перебору в порівнянні з методом повного перебору та є ефективним для визначення показнику нелінійності.

3. Було експериментально досліджено, що використовуючи запропонований метод, можна достатньо точно визначити показник нелінійності. Водночас при збільшенні кількості змінних, від яких залежать нелінійні булеві функції, значення величини помилки значно зменшується, тому розроблений метод може використовуватися для проведення тестування сучасних та новітніх криптографічних алгоритмів захисту даних.

РОЗДІЛ 3

ТЕСТУВАННЯ КРИПТОГРАФІЧНИХ АЛГОРИТМІВ НА ОСНОВІ БУЛЕВИХ ПЕРЕТВОРЕНЬ ВИКОНАННЯМ ОЦІНКИ ЇХ ВЛАСТИВОСТЕЙ

Функціональною основою значної частини алгоритмів захисту інформації є булеві перетворення [5]. Основною перевагою алгоритмів цього класу є простота реалізації в порівнянні з алгоритмами, заснованими на нерозв'язних задачах теорії чисел. Властивості таких алгоритмів засновані на аналітичній нерозв'язності систем нелінійних булевих рівнянь. Потенційно найбільш небезпечними для таких алгоритмів є злам методами лінійного і диференціального криптографічного аналізу [31]. Стійкість до цих видів зламів повністю визначається спеціальними властивостями булевих функціональних перетворень, що використовуються в алгоритмах криптографічного захисту. Основними такими властивостями є нелінійність і диференціальна ентропія булевих функцій.

3.1. Розробка експрес-методу визначення нелінійності булевих функцій

Обов'язковим елементом тестування булевих функцій, які використовуються в алгоритмах захисту інформації є перевірка їх нелінійності. Визначення нелінійності функції від n змінних вимагає витрат обчислювальних ресурсів пропорційних $2^{2 \cdot m}$. Тому на практиці ця задача зводиться до виконання часткових задач – визначення з певною ймовірністю нижньої і верхньої меж нелінійності. У найпростішому випадку необхідно визначити чи є функція лінійної.

Традиційно [1] це виконується через визначення рангу спеціальної матриці, побудованої на основі $2 \cdot m$ довільних наборів вхідних змінних. Локалізація змінних, які лінійно входять в функцію здійснюється шляхом редукування рядків отриманої матриці.

Таким чином, для визначення лінійної залежності у відомому методі [4] використано $2 \cdot m$ наборів і обсяг необхідних обчислень пропорційний $8 \cdot m^3$. Основним недоліком цього підходу, крім значної трудомісткості, є те, що він дозволяє визначити тільки те, що функція лінійно залежить від якоїсь множини змінних. Чисельне визначення нелінійності, як кількісної оцінки стійкості до зламів методами лінійного криптографічного аналізу не досягається.

Якщо врахувати, що на практиці дослідження булевих функцій криптографічних алгоритмів є можливість задання в процесі статистичних досліджень довільних наборів вхідних змінних, можуть бути запропоновані більш ефективні методи визначення лінійності функції, які на відміну від відомих дозволяють швидше виконувати визначення нелінійності.

Нехай m – число вхідних змінних досліджуваної булевої функції. Загальна кількість L лінійних булевих функцій від m змінних дорівнює $2 \cdot (2^m - 1)$. Для великих значень m, які мають місце на практиці, можна вважати, що число L лінійних функцій приблизно дорівнює 2^{m+1}. Позначимо множину всіх лінійних функцій через Θ_0.

На кожному з 2^m можливих наборів значень вхідних змінних рівно половина лінійних функцій приймає значення одиниці. Тому за значенням досліджуваної функції $\beta(b_1,...,b_m)$ на одному з 2^m наборів можна зробити висновок про те, що функція $\beta(b_1,...,b_m)$ не може бути рівною жодній з 2^m лінійних функцій, що утворюють множину Ψ_1 які на цьому наборі приймають протилежне значення при цьому може збігатися з однією з 2^m лінійних функцій, які на даному наборі приймають значення рівне значенню функції і які утворюють множину Θ_1, так, що $\Psi_1 \cup \Theta_1 = \Theta_0$.

За значенням досліджуваної функції $\beta(b_1,...,b_m)$ на іншому наборі B_2 аналогічно можна розділити множину Θ_1 на два рівновеликих підмножини: $\Psi_2 \cup \Theta_2 = \Theta_1$, лінійна функція $\gamma \in \Psi_2$ якщо $\gamma(B_2) \neq \beta(B_2)$ і $\gamma \in \Theta_2$, якщо $\gamma(B_2) = \beta(B_2)$. Аналогічним

чином можна прийти до висновку про те, що аналізом значень функції на $\log_2 L = m+1$ наборах значень змінних можна встановити факт збігу досліджуваної функції з однією з лінійних функцій (в цьому випадку $\Theta_{m+1} \neq \varnothing$). Для того, щоб встановити, що досліджувана функція $\beta(b_1,...,b_m)$ не збігається з жодною з лінійних функцій, тобто є нелінійної, необхідно виконати q порівнянь значень функції $\beta(b_1,...,b_m)$ і однієї лінійної функції, яка складає непорожню множину Θ_q на q наборах вхідних змінних. Це дозволить стверджувати факт лінійності функції $\beta(b_1,...,b_m)$ з ймовірністю $P_q = 1 - 2^{-q}$. Якщо ж множина Θ_q є порожньою, то з цього однозначно випливає, що функція є нелінійною. Якщо задатися апріорі ймовірністю P_q достовірності того, що тестована функція $\beta(b_1,...,b_m)$ буде правильно класифікована як лінійна, для цього необхідно буде K_q значень функції, чисельне значення якого визначається формулою:

$$K_q = \log_2 L + \log_2 \frac{1}{1-P_q} = \log_2 \frac{2^{m-1}}{1-P_q} = m + 1 - \log_2(1 - P_q) \qquad (3.1)$$

Технологічно описаний спосіб визначення нелінійності може бути реалізований попередньою побудовою дерева зміни підмножин Θ, з використанням фіксованих q наборів $B_1,B_2,...,B_q$ вхідних змінних. Це дерево може бути представлено у вигляді функції $\Phi(y_1,\ldots,y_{m+1},\ldots,y_q)$, де $y_1,\ldots,y_q$ – значення функції $\beta(b_1,...,b_m)$ на зазначених вище наборах $B_1,B_2,...,B_q$ і яка приймає значення одиниці, якщо тестована функція $\beta(b_1,...,b_m)$ лінійна і значення нуля – якщо тестована функція – нелінійна, тобто з нелінійності $\beta(b_1,...,b_m)$ з ймовірністю P_q отримуємо $\Phi(\beta(B_1),\beta(B_2),\ldots,\beta(B_s))=0$. Сама функція функції $\Phi(y_1,\ldots,y_{m+1},\ldots,y_q)$ представляється аналітично у вигляді АНФ, а множина наборів $B_1,B_2,...,B_q$ вхідних змінних вибирається таким чином, щоб забезпечити максимальну простоту аналітичного представлення $\Phi(y_1,\ldots,y_{m+1},\ldots,y_q)$.

Якщо число m змінних велике, описаний вище метод визначення лінійності функції може бути технологічно реалізований по-іншому. Функція $\Phi(y_1,\ldots,y_{m+1},\ldots,y_q)$ рівно на 2^{m+1} з 2^q наборах змінних приймає значення одиниці.

Відповідні набори можуть бути впорядковані у вигляді масиву R 2^{m+1} q-розрядних двійкових чисел. При тестуванні на лінійність булевої функції $\beta(b_1,...,b_m)$ обчислюються її значення на q фіксованих наборах, які утворюють q-розрядне двійкове число δ. Якщо δ збігається з одним з чисел масиву R, то функція $\beta(b_1,...,b_m)$ з ймовірністю P_q є лінійної. Таким чином, завдання визначення лінійності булевої функції зводиться до пошуку в упорядкованому масиві, що містить 2^{m+1} чисел. При використанні дихотомічного пошуку або B-дерев реалізація визначення лінійності потребує m+1 цикл звернення до масиву. Таким чином, наведене технологічне рішення запропонованого методу визначення лінійності булевої функції дозволяє вирішувати цю задачу за час, пропорційне числу змінних – m.

3.2. Розробка методики прискореної оцінки нижньої границі нелінійності булевих функцій

Запропонований в попередньому підпункті метод може бути розширений для вирішення іншої важливої задачі – визначення нижньої границі нелінійності. При цьому також використовується аналіз значень функції $\beta(b_1,...,b_m)$ на $s>m+1$ наборах $B_1,B_2,...,B_q$ вхідних змінних $b_1,...,b_m$ з 2^m можливих. За значенням досліджуваної функції $\beta(b_1,...,b_m)$ на першому з цих наборів множина Θ_0 з усіх 2^{m+1} лінійних функцій може бути розділена на дві рівновеликі підмножини: Λ_{10} – до якої входять лінійні функції γ, які на першому наборі не співпадають зі значенням досліджуваної функції і підмножина Λ_{11} – яка об'єднує лінійні функції γ, що співпадають на першому наборі з $\beta(b_1,...,b_m)$, $\gamma_j \in \Lambda_{10}$: $\gamma_j(B_1) \neq \beta(B_1)$, $\gamma_z \in \Lambda_{10}$: $\gamma_z(B_1)=\beta(B_1)$, де $j,z \in \{1,\ldots,2^{m+1}\}$. За результатами аналізу значень функції $\beta(b_1,...,b_m)$ і лінійних функцій на другому наборі B_2 можна розділити множину лінійних функцій Θ_0 на три підмножини: Λ_{20} – до якої входять 2^{m-1} лінійних функцій, що не співпадають з $\beta(b_1,...,b_m)$ на двох наборах, Λ_{21} – яка об'єднує 2^m лінійних функцій, які співпадають зі значенням функції на одному з двох досліджених наборах і підмножина Λ_{22}, що

містить 2^{m-1} лінійних функцій, значення яких співпадають зі значеннями функції $\beta(b_1,...,b_m)$ на обох наборах B_1 B_2.

За результатами порівняння значень досліджуваної функції $\beta(b_1,...,b_m)$ з лінійними функціями на m+1 наборах множину Θ_0 можна буде розділити на m+1 підмножин $\Lambda_{m+1}^1, \Lambda_{m+1}^2, \dots, \Lambda_{m+1}^{m+1}$, причому підмножина Λ_{m+1}^h містить лінійні функції γ, які на наборах $B_1,B_2,...,B_{q+1}$ за значенням співпадають із досліджуваною функцією $\beta(b_1,...,b_m)$ тобто $\gamma_z \in \Lambda_{m+1}^h : \sum_{k=1}^{m+1}\left(1 - f(X_k) \oplus \lambda(X_k)\right) = p$. При цьому кількість лінійних функцій, що потрапляють в кожне з підмножин Λ_{m+1}^z дорівнює C_{m+1}^z.

Міркуючи аналогічним чином можна дійти до висновку про те, що за результатами порівняння значень досліджуваної функції $\beta(b_1,...,b_m)$ з лінійними функціями на s наборах множину Θ_0 можна буде розділити на s підмножин $\Lambda_s^1, \Lambda_s^2, \dots, \Lambda_s^s$, причому підмножини $\Lambda_s^j, j < s - m - 1$ з імовірністю $1\text{-}2^{-(s\text{-}m\text{-}1\text{-}j)}$ будуть порожніми. З того, що значення досліджуваної функції $\beta(b_1,...,b_m)$ на s>m+1 наборах співпадає не більш ніж з q лінійними функціями (q<s-m-1) можна зробити висновок, що з імовірністю $1\text{-}2^{-(s\text{-}m\text{-}1\text{-}q)}$ досліджувана функція має нелінійність не меншу ніж q.

Таким чином, для того, щоб із заданою апріорі ймовірністю P_q зробити висновок про те, що значення нелінійності $NL(\beta)$ булевої функції $\beta(b_1,...,b_m)$, що тестується, є не менше як q, тобто $NL(q) \geq q$ необхідно, визначивши попередньо значення s>m+1+q-$\log_2(1\text{-}P_q)$ виконати порівняння значень функції $\beta(b_1,...,b_m)$ зі значеннями лінійних функцій на s наборах вхідних змінних. При цьому, якщо число виявлених збігів виявиться менше q, то виконується $NL(q) \geq q$.

У технологічному плані заздалегідь можуть бути прораховані кількості співпадаючих лінійних функцій для всіх 2^s варіантів значень, які функція $\beta(b_1,...,b_m)$ може приймати на заздалегідь обраних s наборах вхідних змінних. Ці оцінки можуть бути представлені у вигляді функції $\Phi(y_1,\dots,y_s)$. При цьому функція $\Phi(y_1,\dots,y_s)$ буде приймати значення від 0 до s. Набори змінних $y_1,\dots,y_s$, на яких

функція $\Phi(y_1,\ldots,y_s)$ приймає значення менші ніж q можуть бути оформлені у вигляді постійного упорядкованого масиву s-розрядних чисел. При тестуванні функції $\beta(b_1,...,b_m)$ на заздалегідь обраних наборах $B_1,B_2,...,B_s$ обчислюються значення функції $y_1=\beta(B_1),y_2=\beta(B_2),...,y_s=\beta(B_s)$, формується двійкове число $y_1+y_2\cdot 2+y_3\cdot 2^2+\ldots+y_s\cdot 2^{s-1}$, після чого виконується дихотомічний пошук в упорядкованому масиву, який вимагає не більше s звернень до масиву. Якщо обчислене число буде знайдено в масиві, то нелінійність тестованої функції менше q, тобто $NL(q)\leq q$.

Наприклад, при $m = 4$ для визначення того, що булева функція має нелінійність не менше 3-х з ймовірністю 0.9, число тестових наборів дорівнює 12. Об’єм кластера – близько 2^{11}, число звернень до пам'яті для тестування того, що функція має нелінійність не меншу 3-х дорівнює 11. Якщо в якості тестових вибрати набори значень змінних які утворюють безліч {0001 0010, 0011, 0100, 0101, 0111, 1000, 1001, 1010, 1100, 1110, 1111}, то кластер лінійних функцій буде утворений такою множиною чисел {63, 455, 504, 729, 742, 798, 801, 1108, 1131, 1427, 1452, 1 677, 1 714, 1866, 1909, 2186, 2229, 2381, 2418, 2643, 2668, 2964, 2987, 3294 , 3297, 3353, 3366, 3591, 3640}.

3.3. Розробка алгоритму статистичної оцінки нелінійності булевої функції підбором її лінійної апроксимації

Вага Геммінга булевої функції $\beta(b_1,...,b_m)$ від m змінних – це кількість наборів вхідних змінних $B=\{b_1,\ldots,b_m\}$, на яких функція $\beta(b_1,...,b_m)$ приймає значення одиниці (кількість одиниць в таблиці істинності функції): $HmW(\beta)=\sum_{B\in G}\beta(B)$, де G – множина всіх 2^m можливих наборів значень m змінних.

Булева функція $\beta(b_1,...,b_m)$ від m змінних є балансною, якщо її вага Геммінга дорівнює: $HmW(\beta) = 2^{m-1}$.

Класичний підхід до визначення нелінійності базується на знаходженні оптимального лінійного наближення досліджуваної булевої функції. Кількісне значення нелінійності є відстанню Геммінга до цієї апроксимації ($\gamma^{opt}(B)$):

$$NL(\beta) = HmD(f(B), \gamma^{\text{opt}}(B)) \quad (3.3)$$

Знаходження $\gamma^{opt}(B)$ пов'язане з рядом труднощів обчислювального характеру. Це завдання, в загальному випадку, вимагає повного перебору всіх лінійних функцій $\gamma_j(B)$, $j = 1...2^m$. На сьогодні існує ряд методів визначення нелінійності, які не вимагають повного перебору для функцій вузького класу [17]. Однак для довільних булевих функцій немає методів, що дозволяють відмовитися від повного або скороченого перебору. Таким чином, завдання визначення нелінійності має експоненціальну складність, в залежності від числа змінних m, а, отже, вимагає великих витрат часу при великих значеннях m.

Під множиною оптимальних лінійних наближень деякої функції $\beta(B)$ будемо розуміти таку множину лінійних функцій $\Omega = \{\gamma^{opt}_1,...,\gamma^{opt}_s\}$ для функцій якої виконуються наступне:

$$HmD\left(\beta(B), \gamma_u^{opt}(B)\right) = HmD\left(\beta(B), \gamma_v^{opt}(B)\right), m, n = 1 \dots s \quad (3.4)$$

$$HmD\left(\beta(B), \gamma_u^{opt}(B)\right) = \min_{\gamma(B) \in L} HmD(\beta(B), \gamma(\mathrm{B}))$$

де L – множина всіх лінійних функцій;

$HmD(a(B), b(B))$ – відстань Геммінга між функціями $a(B)$ та $b(B)$.

Виходячи з (3.3) нелінійність булевої функції $\beta(B)$ можна визначити як відстань Геммінга до будь-якої з функцій множини Ω:

$$NL(\beta) = HmD\left(\beta(B), \gamma_u^{opt}(B)\right), u = 1 \dots s$$

Визначимо обмежену множину лінійних функцій $M = \{\tilde{\gamma}_1,\dots,\tilde{\gamma}_t\}$ як таку для функцій якої виконуються такі співвідношення:

$$HmD\left(\beta(B), \gamma_u^{\sim}(B)\right) - HmD\left(\beta(B), \gamma_v^{opt}(B)\right) = \sigma_u, u = 1 \dots t, v = 1 \dots s$$

де σ_u – деяке позитивне ціле число. Введемо позначення:

$$NL^{\sim}(\beta) = \min_{u} HmD\left(\beta(B), \lambda_u^{\sim}(B)\right), u = 1 \dots t$$

Отже, вираз (3.4) можна записати у вигляді:

$$NL^{\sim}(\beta) - NL(\beta) = \sigma_{min} \quad (3.5)$$

де $\sigma_{min} = \min \sigma_u$, $u = 1 \dots t$

Значення $NL^{\sim}(\beta)$ є деякою завищеною оцінкою $NL(\beta)$. При цьому σ_{min} є абсолютним відхиленням нелінійності $NL^{\sim}(\beta)$. Тоді вірно:

$$N(\beta) = N^{\sim}(\beta) - \xi N(\beta)$$

де $\xi = \sigma_{min} / N(\beta)$ – відносне відхилення.

При рівномірно випадковому заповненні множини M лінійними функціями справедливим є вираз:

$$\xi \to 0, \text{ при } t \to 2^m$$

Для визначення нелінійності необхідно провести оцінку через $NL^{\sim}(\beta)$ з мінімально можливим значенням ξ. Так для малого числа змінних можна включити до множини M всі лінійні функції ($M \equiv L$) при цьому, згідно з (3.1), $\xi=0$. Однак при більшій кількості змінних кількість можливих лінійних функцій експоненціально зростає, отже доводиться йти на компроміс між мінімальним ξ і t – кількістю функцій в множині M. При цьому важливо щоб до множини потрапили функції, які є найбільш близькими до $\beta(B)$, тому варіант випадкового заповнення множини M є неприйнятним, з огляду на те, що при цьому велика ймовірність попадання в цю множину функцій далеких від $\beta(B)$. Нижче пропонується метод пошуку лінійних апроксимацій для довільної булевої функції $\beta(B)$ на основі довільної вибірки значень досліджуваної функції. Заповнення множини M з апроксимацій, отриманих за допомогою даного методу, з великою ймовірністю дозволяє досягти меншого значення ξ в порівнянні з випадковим заповненням множини M при однаковій кількості використовуваних функцій t.

У методі використовується довільна вибірка зі значень $Z = \{z_1,...,z_k\}$ функції $\beta(B)$ на k наборах вхідних змінних $B=\{b_1,\dots,b_m\}$:

$$\beta(B_j) = z_j,\ j = 1...k,\ z_j = \{0,1\}$$

На основі цієї вибірки складається матриця ймовірностей P розмірності (m-1) × (m-1) × 4 наведена на рис. 3.1.

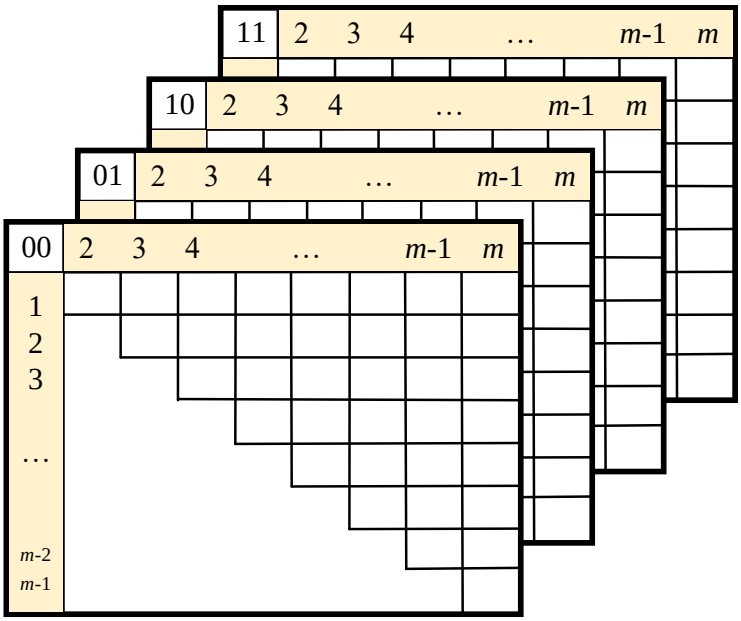

Рис. 3.1. Структура матриці ймовірностей

Кожна з комірок матриці містить:

– $p[u,v,00]$ – ймовірність того, що на довільно обраному наборі значень змінних функція $\beta(B)$ приймає значення одиниці:

$$p[u,v,00] = HmW(\beta(B))/2^m \approx \sum_{B_r \in B} \beta(B_r)/k$$

де $HmW(\beta(B))$ – вага Геммінга булевої функції $\beta(B)$.

– $p[u,v,01]$ – ймовірність того, що на довільно обраному наборі значень змінних функція $(\beta(B) \oplus b_v)$ приймає значення одиниці:

$$p[u,v,01] = HmW(\beta(B) \oplus b_v)/2^m \approx \sum_{B_r \in B} (\beta(B_r) \oplus b_v)/k$$

– $p[u,v,10]$ – ймовірність того, що на довільно обраному наборі значень змінних функція $(\beta(B) \oplus b_u)$ приймає значення одиниці:

$$p[u,v,10] = HmW(\beta(B) \oplus b_u)/2^m \approx \sum_{B_r \in B} (\beta(B_r) \oplus b_u)/k$$

– $p[u,v,11]$ – ймовірність того, що на довільно обраному наборі значень змінних функція $(\beta(B) \oplus b_v \oplus b_u)$ приймає значення одиниці:

$$p[u,v,11] = HmW(\beta(B) \oplus b_v \oplus b_u)/2^n \approx \sum_{B_r \in B} (\beta(B_r) \oplus b_v \oplus b_u)/k$$

Скористаємося такими позначеннями:

BAV – двійковий вектор апроксимації. Елемент вектору BAV[u], встановлений в «1» означає, що у лінійній функції-апроксимації присутній доданок x_i, «0» – відсутність відповідної змінної.

PrM – множина опрацьованих змінних (як тільки опрацьовані всі змінні алгоритм закінчує роботу).

HighBit(r) – старший біт двухбітового значення r.

LowBit(r) – молодший біт двухбітового значення r.

Введемо коефіцієнт «значущості» змінної significance як суму ймовірностей:

$$\text{significance}(\text{PrM}, \text{BAV}, b_e = \text{C}) = \sum_{B_a \in PrM} p[e, a, C.BAV[a]]$$

де «.» – конкатенація двійкових значень;

C – константа C={0,1}.

Коефіцієнт «значущості» є прямо пропорційним до ймовірності того, що на довільно обраному наборі значень змінних сума функції $\beta(B)$ і апроксимації, яка визначається вектором BAV, приймає значення одиниці. Отже, чим менше цей коефіцієнт, тим менше відстань Геммінга між функцією $\beta(B)$ і апроксимацією.

Таким чином, пропонується такий алгоритм знаходження лінійної апроксимації для початкового елемента $q[u,v,w]$:

1. Заповнити нулями вектор апроксимації BAV.
2. PrM = PrM $\cup b_u \cup b_v$, BAV[u] = HighBit(w), BAV[v] = LowBit(w).

Виконати вибір змінної $b_c \notin$ PrM та її значення $g = \{0,1\}$ таким чином, щоб коефіцієнт significance(PrM, BAV, b_c=g) був мінімальний.

3. PrM = PrM $\cup b_c$, BAV[c]=g.
4. Якщо PrM $\neq \{b_1,b_2,\ldots,b_m\}$, то перейти до пункту 2.

В результаті виконання запропонованого алгоритму формується наступна лінійна апроксимація досліджуваної функції:

$$\gamma^{\sim}(B) = BAV[1] \cdot b_1 \oplus BAV[2] \cdot b_2 \oplus \ldots \oplus BAV[m] \cdot x_m$$

Оскільки найкращою апроксимацією може бути функція, до складу якої входить константний доданок «1», то для пошуку такої апроксимації необхідно для кожного з елементів матриці P виконати наступну операцію, яка відповідає інверсії функції:

$$q[u,v,w] = 1 - q[u,v,w],\ u=1\ldots(m\text{-}1),\ v=1\ldots(m\text{-}1),\ w=0\ldots3$$

Після цього необхідно повторити виконання наведеного вище алгоритму, а до отриманої апроксимації додати «1»:

$$\gamma^{\sim}(B) = BAV[1] \cdot b_1 \oplus BAV[2] \cdot b_2 \oplus \ldots \oplus BAV[m] \cdot b_m \oplus 1$$

Вибір початкового елемента матриці $q[u,v,w]$ дуже впливає на апроксимацію, яку отримуємо. Формування множини функцій M пропонується здійснювати на основі виконання наведеного вище алгоритму для різних початкових елементів $q[u_1,v_1,w_1],\ldots,\ q[u_r,v_r,w_r]$. При цьому вибір початкових елементів пропонується здійснювати відповідно до однієї з трьох стратегій:

1. Випадкова – рівномірно розподілений випадковий вибір t початкових елементів матриці P.
2. Мінімальна – вибір t елементів матриці P з мінімальними значеннями.
3. Повна – перебір всіх елементів матриці P. При цьому отримаємо таку кількість функцій в множині M (без урахування можливих повторень апроксимацій):

$$t = (m-1) \cdot (m-1) \cdot 4/2 = 2 \cdot (m-1)^2$$

Таким чином, навіть при використанні «повної» стратегії кількість функцій в M зростає не експоненціально, а поліноміальної (квадратично). Отже, обчислювальна складність методу є поліноміальною.

Для наочної оцінки ефективності запропонованого методу оцінки нелінійності булевої функції в табл. 3.1. наведені дані експериментальних досліджень середньої ξ_{avg} і максимальної ξ_{max} похибок визначення нелінійності для різних функцій при різних значеннях числа змінних для вибірки, яка дорівнює 200, а в табл. 3.2. - для вдвічі більшої вибірки – 400.

Таблиця 3.1.

Результати експериментального дослідження знаходження лінійної апроксимації функції при розмірі вибірки – 200

Кількість змінних	***m*=8**	***m*=9**	***m*=10**	***m*=11**	***m*=12**
Рівномірно-випадкове заповнення множини M					
ξ_{avg}, %	26.923	29.730	27.826	29.852	27.090
ξ_{max}, %	8.500	10.616	13.192	15.181	17.057
Запропонований метод з повною стратегією заповнення					
ξ_{avg}, %	4.691	5.089	6.906	9.222	11.208
ξ_{max}, %	21.698	24.537	29.241	28.191	24.350

Як видно з наведених вище таблиць 3.1 і 3.2 запропонований метод демонструє прийнятну точність при порівняно невеликих вибірках вихідної функції $\beta(B)$. У той же час у порівнянні з випадковим заповненням запропонований метод дозволяє знизити середню відносну похибку майже в два рази. При цьому збільшення розміру вибірки призводить до пропорційного зменшення середньої відносної похибки.

Таблиця 3.2.

Результати експериментального дослідження знаходження лінійної апроксимації функції при розмірі вибірки – 400

Кількість змінних	***m*=9**	***m*=10**	***m*=11**	***m*=12**
Рівномірно-випадкове заповнення множини M				
ξ_{avg}, %	17.130	18.261	20.386	18.789
ξ_{max}, %	5.382	7.278	9.010	10.477
Запропонований метод з повною стратегією заповнення				
ξ_{avg}, %	3.221	3.233	4.566	6.174
ξ_{max}, %	15.315	12.826	15.126	16.649

Таким чином, використання запропонованого методу дозволяє підвищити ефективність визначення нелінійності булевих функцій від великої кількості змінних.

3.4. Статистична перевірка відповідності булевих функцій критерію строгого лавинного ефекту

Для статистичного дослідження строгого лавинного ефекту (диференційної ентропії) булевої функції $\beta(b_1,...,b_m)$ необхідно досліджувати зміну її значення при інвертуванні кожної з m змінних, на яких визначена функція. Для цього треба згенерувати h випадкових наборів B_i, i=1,...,h. Позначимо через B_{il} набір значень вхідних змінних, що відрізняються від B_i значенням l-ої змінної: $B_{il} = \{b_1,\ldots,b_l \oplus 1,\ldots,b_m)$, l=1,…,m.

Для кожного i-го набору B_i необхідно обчислити значення функції на цьому наборі – $\beta(B_i)$, а також m значень функції на наборах з інвертованою однієї змінної

– $\beta(B_{i1}),\beta(B_{i2}),\ldots,\beta(B_{im})$. Тоді можна обчислити статистичну частоту SF_l вимірювання функції по m-ій змінній у вигляді:

$$SF_l = \frac{\sum_{i=1}^{h} \beta(B_i) \oplus \beta(BX_{ih})}{l}$$

Гіпотеза про те, що функція $\beta(b_1,...,b_m)$ відповідає критерію строгого лавинного ефекту по i-тій змінній полягає в тому, що при зміні цієї змінної значення функції змінюється з ймовірністю 0.5. Це означає, що при виконанні гіпотези величина $d_l = 2 \cdot (SF_l - 0.5) \cdot \sqrt{h}$ повинна мати стандартний нормальний розподіл і, відповідно ймовірність того, що функція не відповідає критерію строгого лавинного ефекту по l-ій змінній становить $\Phi(|D_l|)$, де $\Phi(\ )$ – функція Лапласа.

Загальна статистична оцінка ймовірності того, що досліджувана булева функція $\beta(b_1,...,b_m)$ має лавинний ефект за всіма m змінним може бути виконана з використанням критерію χ^2 стосовно вибірці, утвореної значеннями $d_1,d_2,\ldots,d_m$ [5].

Запропоновані методи статистичної оцінки значення нелінійності і відповідності критерію строгого лавинного ефекту булевих функцій засновані на можливості довільного вибору наборів змінних і відповідних значень функції, що складають статистичну вибірку, що часто зустрічається на практиці тестування і сертифікації криптографічних алгоритмів. Це принципово відрізняє їх від методів, основою яких є випадковий характер вибірки, по якій проводиться аналіз нелінійності. Реалізація довільного формування вибірки дозволила розробити методи оцінки нелінійності, що відрізняються від відомих як меншими витратами обчислювальних ресурсів, так і розширеними функціональними можливостями. Це дозволяє в межах наявних обчислювальних ресурсів з використанням запропонованих методів отримати істотно більшу достовірність статистичного аналізу нелінійності та лавинного ефекту в порівнянні з відомими методами.

Висновки до розділу 3

У рамках третього розділу магістерської дисертації було виконано ряд досліджень, які спрямовані на розробку методів і засобів прискорення виконання тестів для оцінки рівня захищеності даних криптографічними алгоритмами в комп'ютерних системах та мережах. Можна зробити такі висновки:

1. Для криптографічних алгоритмів захисту інформації, які базуються на ідеї аналітичної нерозв'язності систем нелінійних булевих рівнянь (до цього класу належать практично всі алгоритми симетричного шифрування та хеш-алгоритми) потенційно найбільшу небезпеку становлять лінійний і диференціальний криптографічний аналізи. Оцінка стійкості до цих видів зламів виконується шляхом визначення показнику нелінійності і диференціальної ентропії булевих перетворень.

2. Описано метод експрес-аналізу нелінійності булевих функцій шляхом їх кластеризації за результатами адаптивного тестування на заздалегідь визначених наборах змінних, а також методика застосування цього методу для статистичної оцінки стійкості до лінійного і диференціального аналізів алгоритмів захисту даних, в основі яких лежать булеві функціональні перетворення, використання який забезпечує прискорення тестування алгоритмів захисту інформації.

3. Описано метод для прискореного знаходження наближеної лінійної булевої функції, обчислювальна складність якого пропорційна n^2 (де n - кількість змінних, на яких визначена функція), який забезпечує істотний виграш у часі в порівнянні з відомим алгоритмом вирішення цієї задачі, обчислювальна складність якої пропорційна $2^{2 \cdot n}$.

4. Описані методи статистичної оцінки значення нелінійності булевих функцій засновані на можливості довільного вибору наборів змінних і відповідних значень функції, що складають статистичну вибірку, що часто використовується на практиці під час тестування і сертифікації криптографічних алгоритмів. Це

принципово відрізняє їх від методів, заснованих на випадковому характері вибірки, на якій проводиться аналіз нелінійності. Реалізація довільного формування вибірки дозволила розробити методи оцінки нелінійності, що відрізняються від відомих як меншими витратами обчислювальних ресурсів, так і розширеними функціональними можливостями. Це дозволяє в межах наявних обчислювальних ресурсів отримати з використанням запропонованих методів істотно більшу достовірність статистичного аналізу нелінійності і строгого лавинного ефекту в порівнянні з відомими методами.

РОЗДІЛ 4

ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ ЗАПРОПОНОВАНОГО МЕТОДУ

Для проведення експериментального моделювання роботи розробленого методу, збору статистичних даних, а також проведення тестування новітніх алгоритмів криптографічного захисту даних було розроблено ряд програмних застосунків.

Перед програмними застосунками висувалися такі задачі:

- виконання збору статистичних даних (обсяг необхідної пам’яті, час виконання, точність обчислення) для оцінки ефективності роботи розробленого методу;
- забезпечення засобами для виконання прискореного визначення показнику нелінійності.

Завдання, які повинна виконувати програма, були такими:

- задання вхідної нелінійної булевої функції, залежної від n змінних, нелінійність якої необхідно обчислити;
- проведення перевірки таких параметрів вхідної нелінійної булевої функції, як розмір та збалансованість. Виведення повідомлення про невідповідність вимогам до вхідної булевої функції за умови некоректності цих параметрів;
- виведення множини лінійних функцій, значення відстані Геммінга між якими та вхідною нелінійною булевою функцією є мінімальними, отриманої в результаті роботи розробленого алгоритму, а також підсумкового значення нелінійності;
- надання можливості поетапного виведення проміжних значень під час роботи розробленого алгоритму;

- проведення збору статистичних даних про середню кількість виконаних операцій перебору, яка була необхідна для обрахунку показнику нелінійності нелінійних булевих функцій від заданої користувачем кількості змінних *n*. Нелінійні булеві функції для проведення статистичного дослідження генеруються програмно, їхня кількість задається користувачем;
- організація паралельної обробки вхідних нелінійних булевих функцій для збору статистики з метою досягнення зменшення часу роботи програми та ефективного завантаження ядер процесору.

Програмний додаток було реалізовано за допомогою мови програмування *Rust*. Мова *Rust* є сучасною системною високорівневою мовою програмування, яка була обрана з огляду на те, що:

- мова програмування *Rust* має потужну систему перевірки правильності програми на момент компіляції, що заснована на строгих правилах щодо володіння об'єктами. Як наслідок, це дає можливість відловити значно більшу кількість логічних помилок, аніж при використанні інших компілюємих мов програмування;
- не дивлячись на те, що мова програмування *Rust* є високорівневою мовою, вона дозволяє генерувати оптимізовані низькорівневі бінарники, які не поступають у швидкості тим, що генеруються за допомогою компіляторів для мов *C* та *C++*. Таким чином, це дає можливість отримати велику швидкодію програми і зменшити час її виконання, що є важливим фактором при перевірці запропонованого методу для великої кількості змінних;
- мова програмування *Rust* є високорівневою мовою програмування, що дозволяє використовувати високорівневі концепції та структури у поєднанні з функціональною парадигмою програмування, отримуючи

таким чином зрозумілий короткий код, який легко читається та розширюється;

– *Rust* є сучасною мовою програмування, яка активно розробляється та має велику користувацьку базу та чудову документацію, що дозволяє легко знайти відповіді на будь-які запитання при розробці проекту.

4.1. Основні компоненти програми

Програмний продукт складається з файлу *main.rc*, у якому прописана логіка виконання запропонованого методу, описані структури даних та методи для його виконання, а також задаються такі вхідні дані для роботи методу, як бітова довжина та кількість нелінійних булевих функцій для проведення статистичного дослідження або конкретна нелінійна булева функція у форматі вектору значень її таблиці істинності для обрахунку показнику її нелінійності.

Вхідними даними, які очікуються від користувача, на вибір є:

– для отримання статистичних даних: значення кількості змінних PARAMETERS, від яких залежать аналізовані нелінійні збалансовані булеві функції, кількості STATISTIC_COUNT нелінійних булевих функцій, яку необхідно згенерувати для обрахунку статистики, кількості THREAD_COUNT потоків, які будуть створені для паралельної обробки нелінійних булевих функцій, обчислення показнику їх нелінійності та підрахунку виконаних для цього операцій перебору;

– для обрахунку показнику нелінійності: нелінійна збалансована булева функція.

Усі інші параметри програми обчислюються автоматично в залежності від заданих користувачем змінних, які були описані раніше.

Глобальною константною змінною є PARAMETERS, яка вказує від якої кількості змінних повинні залежати згенеровані нелінійні збалансовані булеві

функції. Так, наприклад, для генерації булевих функцій, залежних від чотирьох змінних x_0, x_1, x_2, x_3, значення цього параметру буде дорівнювати 4: PARAMETERS = 4.

Для написання програми було використано такі стандартні типи даних мови *Rust*.

Primitive Type array – структура даних, яка має фіксований розмір та зберігає елементи однакового типу. Цей тип даних є корисним, коли завчасно відомо кількість елементів, які будуть до нього входити. Використання цього типу даних дозволяє більш ефективно використовувати пам’ять. Розмір такого масиву повинен бути вказаний на момент компіляції, а пам’ять для розміщення даних буде виділено на стеці.

Struct std::vec::Vec – структура даних, розмір якої невідомий на момент компіляції та яка зберігає дані однакового типу. Цей тип даних працює з трьома параметрами: вказівником на дані, довжиною та ємністю. Ємність вказує, скільки пам’яті зарезервовано для зберігання елементів вектору. Вектор може збільшуватися поки значення довжини, менше ємності. Коли цей поріг потрібно перевищити, вектор перерозподіляється з більшою ємністю. Пам’ять для розміщення даних виділяється динамічно на купі.

Struct std::string::String – примітивний строковий тип даних для роботи зі строками (виконання операцій присвоєння, зміни строк, порівняння).

Struct std::thread::Thread - структура даних, яка дозволяє виконувати незалежні частини програми одночасно та асинхронно. Таке розбиття обчислень на декілька потоків дозволяє поліпшити продуктивність роботи програми.

Для зручності роботи з даними було введено такі власні типи даних.

struct Function – структура, яка репрезентує булеву функцію у форматі вектору значень таблиці істинності. Ця структура імплементує основні функції для моделювання та аналізу роботи розробленого методу, а саме: функцію точного

обрахунку показнику нелінійності вхідної нелінійної збалансованої булевої функції шляхом виконання повного перебору, функція реалізації запропонованого методу прискореного визначення нелінійності, функції для збору статистичних даних.

У структури *Function* є одне поле *values*, яке має тип *Vec<bool>*, та призначене для зберігання булевої функції у форматі вектору значень таблиці істинності.

Структура *Function* має такі методи:

- *fn new(values: Vec<bool>)* – конструктор, що виконує створення нового об’єкту типу *Function* та ініціалізацію поля *values* шляхом його заповнення вектором значень таблиці істинності, який був отриманий в аргумент *values* цієї функції.
- fn calculate_nonlinearity(&self) – функція точного обчислення показнику нелінійності вхідної нелінійної булевої функції, яке виконується шляхом повного перебору всіх можливих лінійних булевих функцій від заданої кількості змінних. Спочатку формується множина всіх можливих лінійних функцій від заданої кількості змінних. Формування цієї множини виконується таким чином. Кожна лінійна функція може бути представлена у форматі двійкового числа, у якому значення 1 буде означати присутність деякої змінної, а значення 0 – її відсутність. Для виконання описаної операції на вхід функції *get_lambda_function()* передається ціле число, яке після роботи цього методу матиме формат вектору значень таблиці істинності вхідної лінійної функції. Наступним етапом є обрахунок значень відстаней Геміннга між вхідною нелінійною булевою функцією та кожною лінійною булевою функцією з множини усіх можливих та знаходження найменшого значення відстані Геммінга, що є показником нелінійності вхідної нелінійної булевої функції.

- *fn calculate_h(&self, indices: &Vec<u32>)* – функція для обрахунку відстані Геммінга між вхідною нелінійною булевою функцією та лінійною булевою функцією, яку передали в аргумент *indices* цієї функції у форматі вектору індексів змінних, від яких залежить порівнювана лінійна функція. Іншими словами ця функція обчислює значення нелінійності вхідної нелінійної булевої функції.
- *fn calculate_min_h(&self, sorted_stats: [(u32, f64); PARAMETERS as usize])* – функція, яка реалізує роботу запропонованого методу прискореного обчислення значення нелінійності булевих функцій. У результаті своєї роботи ця функція повертає обраховане значення нелінійності та значення кількості операцій перебору, яку для цього необхідно було виконати.
- *fn already_added(selected: &Vec<Vec<u32>>, test: u32)* – допоміжна функція для перевірки того, чи лінійна функція, передана в аргумент *selected*, вже входить до множини ефективних на певному кроці лінійних функцій. Повертає булеве значення результату перевірки.
- *fn get_lambda_function(index: u64)* – функція для отримання лінійної функції у форматі таблиці істинності. На вхід до цієї функції в аргумент *index* передається ціле число, яке першим етапом перетворюється в набір індексів змінних, від яких залежить ця лінійна функція, а другим етапом – у вихідний вектор булевих значень, який представляє таблицю істинності лінійної функції.
- *fn get_base_number(parameter: u32, index: u32)* – допоміжна функція, задачею якої є проінвертувати вхідний набір змінних таблиці істинності, який переданий в аргумент *parameter*, по індексу змінної, вказаному в аргументі *index*. Так, наприклад, вхідне значення 100 буде проінвертоване по змінній x_1 (*index* = 1) і результатом роботи функції буде набір 110.

- *fn get_tuple(base_number: u64, bit_to_invert: u32)* – допоміжна функція, яка створює об’єкт типу *Tuple* та зберігає в нього два вхідні набори змінних таблиці істинності: переданий в аргумент набір змінних *base_number* та набір, проінвертований до набору *base_number* по індексу змінної *bit_to_invert*.
- *fn get_sorted_statistics(&self)* – допоміжна функція, яка виконує два перші пункти запропонованого алгоритму, а саме обраховує та сортує у порядку зменшення масив значень імовірностей того, що інвертування значення певної змінної нелінійної булевої функції, призведе до зміни її значення.
- *fn generate_nonlinear_func(random: &mut FastStdRand)* – допоміжна функція для генерування об’єкту типу *Function* – нелінійної збалансованої булевої функції від кількості змінних, заданої константою PARAMETERS, у форматі вектору значень таблиці істинності.
- *fn get_linear_function(current_lambda_func: &Vec<bool>)* – допоміжна функція перетворення лінійної функції, переданої в аргумент *current_lambda_func*, з формату вектору значень таблиці істинності в алгебраїчну нормальну форму.

Для реалізації способу отримання значення конкретного елементу поля *values* об’єкту типу *Function* по індексу *i*, який передається в якості аргументу, виконується впровадження ознаки *Index* в незмінному контексті для цієї структури: *impl Index<usize> for Function* – за допомогою функції *fn index<'a>(&'a self, i: usize)* якої виконується операція індексування елементів типу *Function*.

struct LinearFunction – структура, що зберігає лінійну булеву функцію у форматі, який буде зручним для подальшої роботи з нею, та визначає допоміжні методи для її опрацювання. Лінійна булева функція зберігається у форматі вектору значень індексів змінних, від якої залежить ця функція. Так, наприклад, для лінійних булевих функцій від чотирьох змінних x_0, x_1, x_2, x_3 та можливої компоненти

заперечення – «1», збереженими індексами їх представлення є: для заперечення («1») – 0, x_0 – 1, x_1 – 2, x_2 – 3, x_3 – 4. Таким чином, АНФ лінійної булевої функції: $1 \oplus x_1 \oplus x_3$ – буде представлено у вигляді вектору індексів {0, 2, 4}.

У структури *LinearFunction* є одне поле *indices*, яке має тип *Vec<u32>*, та призначене для зберігання вектору значень індексів змінних, які входять до складу певної лінійної булевої функції.

Структура *LinearFunction* має такі методи:

- *fn new(mut number: u64)* – конструктор, що виконує створення нового об’єкту типу *LinearFunction* та ініціалізацію поля *indices* шляхом його заповнення вектором значень індексів змінних, які входять до складу певної лінійної булевої функції, яка була передана в аргумент *number* цієї функції.
- fn len(&self) – допоміжна функція для отримання розміру поля *indices* об’єкту типу *LinearFunction*.

Для реалізації способу отримання значення конкретного елементу поля *indices* об’єкту типу *LinearFunction* по індексу *i*, який передається в якості аргументу, виконується впровадження ознак *Index* та *IndexMut* в незмінному та змінному контекстах, відповідно, для цієї структури: *impl Index<usize> for LinearFunction* та *impl IndexMut<usize> for LinearFunction* – за допомогою функцій *fn index<'a>(&'a self, i: usize)* та *fn index_mut<'a>(&'a mut self, i: usize)* яких виконується операція індексування елементів типу *LinearFunction*.

struct Tuple – допоміжна структура для зручності організації обчислення статистичних показників зміни значення вхідної нелінійної булевої функції, коли була інвертована певна *k*-та змінна цієї функції. У поля цієї структури зберігаються значення наборів змінних, які мають один різний *k*-тий розряд (наприклад, набори 01100 та 01110).

У структури *Tuple* є два поля *number1* та *number2*, які мають тип *u64*, та призначені для зберігання значень наборів змінних для порівняння на них значень булевої функції.

Структура *Tuple* має такі методи:

- fn empty() – конструктор, який буде використаний за замовчуванням для створення нового об’єкту типу *Tuple*, при цьому поля *number1* та *number2* будуть проініціалізовані значеннями, рівними 0.
- *fn new(number1: u64, number2: u64)* – конструктор, що виконує створення нового об’єкту типу *Tuple* та ініціалізацію полів *number1* та *number2* шляхом заповнення їх значеннями, які були отримані в аргументи *number1* та *number2* цієї функції, відповідно.

Для реалізації способу отримання значення або поля *number1*, або поля *number2* об’єкту типу *Tuple* по індексу *i*, який передається в якості аргументу, виконується впровадження ознак *Index* та *IndexMut* в незмінному та змінному контекстах, відповідно, для цієї структури: *impl Index<usize> for Tuple* та *impl IndexMut<usize> for Tuple* – за допомогою функцій *fn index<'a>(&'a self, i: usize)* та *fn index_mut<'a>(&'a mut self, i: usize)* яких виконується операція індексування елементів типу *Tuple*. Додатково виконується перевірка: *assert!*(*i <= 1*) – того, що аргумент *i* має одне з допустимих значень: 0 або 1 – при заданні відмінних від цих двох значень буде повідомлятися про помилку запиту значення по індексу, якого не існує.

struct Brace – допоміжна структура, яка застосовується для трансформації лінійної булевої функції з формату вектору значень таблиці істинності до АНФ.

У структури *Brace* є одне поле *elem*, яке має примітивний тип *tuple* (*u32, bool*), та призначене для зберігання пари значень: значення індексу певної змінної лінійної булевої функції та булеве значення, чи є інвертованою ця змінна.

Структура *Brace* має такі методи:

- *fn empty()* – конструктор, який буде використаний за замовчуванням для створення нового об’єкту типу *Brace*, при цьому поле *elem* буде проініціалізоване значенням (0, *false*).
- *fn new(arg: u32, is_inverted: bool)* – конструктор, що виконує створення нового об’єкту типу *Brace* та ініціалізацію поля *elem* шляхом його заповнення значеннями, які були отримані в аргументи *arg* та *inverted* цієї функції, відповідно.
- *fn index(&self)* – допоміжна функція для зручного отримання першого значення *arg*, яке має тип *u32*, поля *elem* об’єкту *Brace*.
- *fn inverted(&self)* – допоміжна функція для зручного отримання другого значення *inverted*, яке має тип *bool*, поля *elem* об’єкту *Brace*.

4.2. Статистичне моделювання запропонованого методу

У цьому підпункті буде детально розглянуто основну логіку роботи програми, яка описана у функції *main*.

На початку функції відбувається оголошення декількох констант, якими регулюється подальша робота програми. Таких констант дві:

- STATISTIC_COUNT – константа, яка задає кількість елементів вибірки для статистичного аналізу. Визначає кількість ітерацій, які будуть виконані для одних і тих самих значень вхідних параметрів. Кожна з ітерацій дасть чергові дані для статистичного аналізу. Таким чином, більше значення цієї константи дає змогу отримати більше даних вибірки, що в свою чергу дає можливість отримати результати з більшою точністю, проте виконання програми займе більше часу.
- THREADS_COUNT – константа, яка задає кількість потоків, що будуть створені та використовуватимуться для статистичного аналізу. Дозволяє пришвидшити виконання програми за рахунок розпаралелювання

незалежних ітерацій. Для максимального зменшення часу виконання програми рекомендується визначити значення цієї константи рівним кількості ядер у системі, за умови, що вони не зайняті іншими «важкими» задачами у системі.

Наступним кроком є оголошення двох векторів заздалегідь відомого розміру, що будуть використовуватися для зберігання даних статистичного аналізу, а саме *passes_stats*, який буде накопичувати кількість проходів, та *percent_stats*, який буде накопичувати похибки. Обидва вектори мають розмір STATISTIC_COUNT і пам’ять під них виділяється на початку лише один раз.

Мова програмування *Rust* була розроблена в першу чергу як системна мова програмування. У ній дуже якісно та ретельно реалізовані аспекти, що відповідають за потоки та паралелізм. Коректність володіння даними, що є основною при паралельному програмуванні, перевіряється ще на етапі компіляції.

Для того, аби мати можливість працювати з одними і тими самими даними – векторами для зберігання статистики у даному випадку – необхідно використати м’ютекс, який буде «охороняти» дані від одночасної зміни зі сторони декількох потоків. Це робиться таким чином:

let mutex = Arc::new(Mutex::new((percent_stats, passes_stats)));

До м’ютекса зберігається кортеж, що складається з двох елементів, до яких необхідно забезпечити паралельний доступ.

Наступним кроком є створення вектору потоків, за допомогою якого до них можна буде доступатися для контролю потоків після їх створення.

Далі відбувається безпосередньо створення потоків, а тому робиться цикл з ітератором *thread_id*, що проходить по діапазону чисел від 0 до THREAD_COUNT і відповідає за індекс поточного потоку.

На кожній ітерації цього циклу відбувається така послідовність дій. Спочатку створюється копія головного м’ютексу для того, аби черговий потік зміг отримати

повне володіння цією копією (це робиться згідно вимог, що висуває мова програмування *Rust* для забезпечення коректності роботи багатопотокових програм). Далі створюється та додається до вектору потоків новий потік за допомогою функції *thread::spawn*, що приймає на вхід лямбда-функцію – тіло нового потоку.

Тіло кожного потоку складається з виконання таких кроків. На початку виводиться повідомлення, який потік (з яким індексом *thread_id*) запускається. Після цього створюється власний (для цього потоку) генератор псевдовипадкових послідовностей, що був створений раніше – *FastStdRand*. Наступним кроком оголошується цикл з ітератором *i*, який проходить по значенням від *thread_id* до STATISTIC_COUNT з кроком THREAD_COUNT. Таким чином кожному потоку виділяється рівна частина роботи, яку потрібно виконати. На кожній ітерації цього циклу відбувається створення нової нелінійної функції за допомогою власного генератора псевдовипадкових послідовностей. Оскільки головне завдання програми – перевірити, наскільки якісно запропонований метод може знайти нелінійність, роблячи це набагато швидше, то для оцінки його роботи необхідно знайти справжнє значення нелінійності створеної функції. Це значення знаходиться за допомогою виклику *calculate_nonlinearirty* для новоствореної нелінійної функції. Згідно роботи алгоритму, необхідно згенерувати відсортовану статистику для цієї нелінійної функції. Це робиться викликом *get_sorted_statistics*. На основі знайденої відсортованої статистики виконується пошук самого значення нелінійності (*curr_nonlin*) за допомогою запропонованого методу, а також кількості ітерацій (*curr_n*), що знадобилися для її роботи. Це робиться викликом функції *calculate_min_h*. Неточність знайденої нелінійності обраховується як частка підрахованої за допомогою запропонованого методу нелінійності від справжньої нелінійності, і записується у змінну *percent*. Усі три значення *real_nl*, *curr_nonlin* та *percent* виводяться на екран.

Наступним кроком є накопичення отриманих даних для статистичного аналізу, а тому відбувається захоплення м’ютексу за допомогою виклику *mutex_clone.lock().unwrap()*, який поверне кортеж з двома векторами статистики. Після цього до обох векторів додаються нові значення *percent* та *curr_n*, відповідно. Тепер можна відпустити м’ютекс викликом функції *drop*.

На цьому завершується тіло потокової лямбда-функції, а з ним і зовнішній цикл, що створює потоки.

Далі для кожного потоку (з вектору *threads*) викликається метод *join*, що примушує головний потік програми очікувати на завершення роботи усіх створених потоків.

Наступним кроком отримуються значення векторів статистичного аналізу і починається виведення результатів роботи.

Для цього оголошуються та ініціалізуються нульовим значенням змінні *max_percent*, *sum_percent* та *sum*.

Далі для усіх елементів статистичного аналізу (відповідних елементів масивів *passes_stats* та *percent_stats*) виконується така послідовність кроків: до змінної *sum* додається чергове значення з масиву *passes_stats*, а до *sum_percent* – чергове значення масиву *percent_stats*, яке додатково також виводиться на екран. При цьому, якщо це значення неточності (змінна *percent*) є більшим за поточний максимум, то оновлюється значення *max_percent*.

Після підрахунку у циклі значень цих змінних здійснюється пошук середніх значень *sum* та *sum_percent*, яке зберігається до змінних *mean* та *mean_percent*, відповідно.

Значення *mean*, *mean_percent* та *max_percent* виводяться на екран. Цим завершується робота програми.

4.3. Інструкція користування розробленою програмою

Для визначення показнику нелінійності заданої користувачем вхідної нелінійної булевої функції запропонований алгоритм виконує таку послідовність кроків:

1. Очікує від користувача введення нелінійної збалансованої булевої функції у форматі вектору значень її таблиці істинності. Ця вхідна нелінійна булева функція зберігається у змінну *test_nonlinear_function* для подальшої роботи з нею.
2. Виконує формування відсортованої у порядку зменшення статистики зміни значення вхідної нелінійної булевої функції при виконанні інвертування кожної її змінної. Для цього виконується виклик функції *test_nonlinear_function.get_sorted_statistics()*. Результат роботи цієї функції використовується на наступному етапі роботи алгоритму.
3. Виконує обчислення показнику нелінійності вхідної нелінійної булевої функції за запропонованим методом. Для цього виконується виклик функції *test_nonlinear_function.calculate_min_h(sorted_stats)*, в аргумент якої передається сформована на попередньому кроці статистика.

Додатково користувач моє можливість обрати режим поетапного виведення проміжних результатів роботи розробленого методу, які будуть виведені на екран разом із обрахованим показником нелінійності та кількістю операцій перебору, яку необхідно було виконати у ході роботи методу. Наведемо такий приклад роботи програми:

```
-------------------------------------------
User input: 00001111011011101010000010101011
-------------------------------------------
Calculated by brute force nonlinearity = 10
===========================================
Linear functions with min Hamming distance
-------------------------------------------
x4 + 1
-------------------------------------------
```

```
x2 + x0
-------------------------------------------
x2 + x1 + x0
===========================================
Sorted calculated values of probabilities
-------------------------------------------
x4 = 0.625
x0 = 0.5
x2 = 0.5
x3 = 0.375
x1 = 0.25
-------------------------------------------
sorted_stats = [ x4 x0 x2 x3 x1 ]
===========================================
v = 0, q = 1 lin_func_0 = x4
-------------------------------------------
v = 1
-------------------------------------------
w = 0 c = 0
HmD ( x4 ) = 10
HmD ( x4 x0 ) = 12
lin_func_0 = x4
lin_func_1 = x0
q = 1
-------------------------------------------
v = 2
-------------------------------------------
w = 0 c = 0
HmD ( x4 ) = 10
HmD ( x4 x2 ) = 14
lin_func_0 = x4
lin_func_1 = x0
lin_func_2 = x2
q = 1
-------------------------------------------
w = 1 c = 1
HmD ( x0 ) = 14
HmD ( x0 x2 ) = 10
lin_func_0 = x4
lin_func_1 = x0 x2
lin_func_2 = x2
q = 1
-------------------------------------------
v = 3
-------------------------------------------
w = 0 c = 0
HmD ( x4 ) = 10
```

```
HmD ( x4 x3 ) = 14
lin_func_0 = x4
lin_func_1 = x0 x2
lin_func_2 = x2
lin_func_3 = x3
q = 1
-------------------------------------------
w = 1 c = 1
HmD ( x0 x2 ) = 10
HmD ( x0 x2 x3 ) = 14
lin_func_0 = x4
lin_func_1 = x0 x2
lin_func_2 = x2
lin_func_3 = x3
q = 1
-------------------------------------------
w = 2 c = 1
HmD ( x2 ) = 12
HmD ( x2 x3 ) = 16
lin_func_0 = x4
lin_func_1 = x0 x2
lin_func_2 = x2
lin_func_3 = x3
q = 1
-------------------------------------------
v = 4
-------------------------------------------
w = 0 c = 0
HmD ( x4 ) = 10
HmD ( x4 x1 ) = 14
lin_func_0 = x4
lin_func_1 = x0 x2
lin_func_2 = x2
lin_func_3 = x3
lin_func_4 = x1
q = 1
-------------------------------------------
w = 1 c = 1
HmD ( x0 x2 ) = 10
HmD ( x0 x2 x1 ) = 10
lin_func_0 = x4
lin_func_1 = x0 x2 x1
lin_func_2 = x2
lin_func_3 = x3
lin_func_4 = x1
q = 1
-------------------------------------------
```

```
w = 2 c = 1
HmD ( x2 ) = 12
HmD ( x2 x1 ) = 16
lin_func_0 = x4
lin_func_1 = x0 x2 x1
lin_func_2 = x2
lin_func_3 = x3
lin_func_4 = x1
q = 1
-------------------------------------------
w = 3 c = 1
HmD ( x3 ) = 16
HmD ( x3 x1 ) = 16
lin_func_0 = x4
lin_func_1 = x0 x2 x1
lin_func_2 = x2
lin_func_3 = x3 x1
lin_func_4 = x1
q = 1
===========================================
Result set of linear functions
-------------------------------------------
lin_func_0 = x4
lin_func_1 = x0 x2 x1
lin_func_2 = x2
lin_func_3 = x3 x1
lin_func_4 = x1
-------------------------------------------
passes_made = 20
```

Наведений приклад демонструє, що на початку своєї роботи програма виконує обчислення точного значення показнику нелінійності вхідної нелінійної булевої функції шляхом виконання повного перебору усіх можливих лінійних булевих функцій від заданої кількості змінних, перевіряючи такі її параметри, як розмір та збалансованість. Далі здійснюється виведення множини лінійних булевих функцій, відстань Геммінга між якими та вхідною нелінійною булевою функцією є мінімальною та дорівнює обрахованому значенню нелінійності.

Наступним етапом є виконання пошуку лінійних булевих функцій за розробленим алгоритмом. У виводі можна побачити відсортований у порядку зменшення масив імовірностей зміни значення вхідної булевої функції при

виконанні інверсії значень певної змінної, а нижче – проміжні значення допоміжних індексів, визначені значення відстаней Геммінга та лінійні функції, які алгоритм відбирає для своєї роботи на кожному кроці.

У результаті роботи методу було отримано лінійні булеві функції *lin_func_0*, *lin_func_1*, *lin_func_2*, *lin_func_3*, *lin_func_4* та кількість виконаних операцій перебору *passes_made*, яка дорівнює 20: *passes_made* = 20.

Отримана шляхом динамічного формування лінійна булева функція *lin_func_1* збігається з третьою лінійною функцією: *lin_func_1* = $x_0 \oplus x_2 \oplus x_1$ – з множини найближчих, у сенсі Геммінга, лінійних функцій, визначеною методом повного перебору.

Висновки до розділу 4

У четвертому розділі магістерської дисертації детально розглянуто програмну реалізацію експериментального моделювання та практичного використання запропонованого методу, підсумовуючи які можна зробити такі висновки:

1. Було розроблено низку програмних застосунків на мові програмування Rust для проведення моделювання, збору статистики та тестування криптографічних алгоритмів захисту даних, в основу яких покладено розроблений метод прискореного визначення показнику нелінійності булевих перетворень.

2. Результати експериментального статистичного моделювання роботи розробленого алгоритму для тестування нелінійності збалансованих булевих функцій, залежних від різної кількості змінних, доводять ефективність запропонованого методу в порівнянні з методом точного визначення нелінійності та відомими алгоритмами прискореного визначення нелінійності. Для нелінійних булевих функцій, залежних від 12 і більше змінних, що використовуються у сучасних криптографічних механізмах захисту інформації, обрахунок показнику нелінійності виконується особливо помітно швидше та є достатньо акуратним.

3. З метою ефективнішого використання обчислювальних ресурсів та зменшення часу виконання експериментального статистичного дослідження були використані механізми мови Rust для організації паралельної обробки нелінійних булевих функцій. В залежності від наявних умов можна вказати значення параметрів кількості потоків та кількості нелінійних булевих функцій, які будуть приймати участь у підрахунку статистики.

ВИСНОВКИ

За результатами виконання магістерської дисертації, направленої на вирішення наукової задачі прискорення тестування рівня захищеності широкого класу криптографічних алгоритмів, що мають за основу булеві перетворення, можна зробити наступні висновки:

1. Проведений оглядовий аналіз існуючих статистичних методів тестування нелінійності булевих перетворень показав, що вони в силу свого статистичного характеру не в повній мірі відповідають вимогам, які висуває розробка сучасних криптографічних алгоритмів захисту даних. Оскільки в останні роки набувають популярності системи автоматичної побудови криптографічних алгоритмів, етапами яких є генерація булевих перетворень та визначення їх криптографічної стійкості, а також те, що кількість змінних булевих функцій наразі є достатньо великою, то це відповідно призводить до значного зростання затрат часу та технічних ресурсів для обчислення нелінійності.

2. Теоретично обґрунтовано, розроблено та досліджено метод прискореного визначення нелінійності булевих перетворень, який відрізняється від відомих використанням динамічного програмування для побудови лінійних апроксимацій, за рахунок чого досягнуто прискорення визначення нелінійності булевих перетворень від великої кількості змінних.

3. Розроблено програмні засоби для практичної реалізації розробленого методу прискореного визначення нелінійності булевих перетворень для криптографічних застосувань.

4. Теоретично та експериментально, з використанням розроблених програних засобів, показано, що обчислювальна складність запропонованого методу носить квадратичний характер: $O(n^2)$. Це дозволяє значно скоротити час визначення нелінійності у порівнянні з відомими методами та підвищити надійність отриманих в результаті застосування методу оцінок. Величина похибки визначення

нелінійності при $n>14$ в запропонованому методі гарантовано менше, ніж у відомих. А при $n=16$ похибка не перевищує 0.8%.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Давиденко А.Н. Вероятностная оценка надежности реализации функций защиты информации // Моделювання та інформаційні технології: Зб.наук. праць.-Львів:НВМ ПТ УАТ. – 2002. – Вип.14. – С.64-70.

2. Тилборг ван Х. К.А. Основы криптологии / Тилборг ван Х. К. А. – М.:Мир, 2006ю – 471с.

3. Основы криптографии / [Алферов А. П., Зубов А. Ю., Кузьмин А. С., Черемушкин А. В.]. – М.:«Гелиос АРВ», 2001. – 480 с.

4. Барабаш А. В. Криптография (аспекты защиты) / Барабаш А. В., Шанкин Г. П. – М.:СОЛОМОН-Р, 2002. – 512 с.

5. Ленков С. В. Методы и средства защиты информации / Ленков С. В., Перегудов Д. А., Хорошко В. А. – К.:АРИЙ, 2008, Том I – 464 с., Том II – 344 с.

6. Марковский А.П., Абу Усбах А.Н., Иваненко Я.П. К вопросу об определении нелинейности булевых функций специальных классов. // Вісник Національного технічного університету України ”КПІ” Інформатика, управління та обчислювальна техніка. – 2002. – № 37. – С.14-24.

7. Марковський О.П. Комбинаторный анализ булевых функций специальных классов для систем криптографической защиты информации // А.П. Марковский, Э.Р. Искаков, Г.В. Гарасимович // Збірник доповідей міжнародної науково-технічної конференції “The International Conference on Security, Fault Tolerance, Intelligence” (ICSFTI2018). – Київ, 10-11 травня 2018. – С.42-50.

8. Мустафа Акрам Ареф Найеф. Статистическая оценка нелинейности и лавинного эффекта булевых функций. // Вісник Національного технічного університету України ”КПІ”. Інформатика, управління та обчислювальна техніка. – 2002. – № 38. – С.106-113.

9. Самофалов К.Г., Ель-Хами И., Кожемякин С.В. Использование апарата булевих функций для оценки эффективности криптографических алгоритмов защиты информации. // Збірник статей “Правове, нормативне та метрологічне забезпечення системи захисту інформації в Україні” – К.:Вид-во ЕКМО, 2000. – С.244-250.

10. Mesnager S. Bent Function: Fundamentals and Results / S. Mesnager // IEEE Trans. on Information Theory. – 2016. – Vol.62, No. 7. – pp. 1825-1834.

11. Gutierrez J., Shparlinski I., Winterhof A. On the Linear and Nonlinear Complexity Profile on Nonlinear Pseudorandom Number Generators // IEEE Trans. Information Theory. – 2003. – Vol. 49. – № 1. – P.60-64.

12. Matsui M. Linear cryptanalysis method for DES cipher. // Proceeding of Eurocrypt-93, LNCS 765. Springer, – 1994. – P.386-397.

13. Zhang M. Maximum Correlation Analysis of Nonlinear Combining Function in Stream Cipher. // Journal of Cryptology, – 2000. – Vol.13. – P. 301-313.

14. Zhang X., Zheng Y. On Relationship among Avalanche, Nonlinearity and Correlation Immunity. // Proceedings of Asiacrypt-2000, LNCS 1976, Springer, – 2000. – P.135-142.

15. Xiang C. A construction of linear codes from Boolean functions / C. Xiang, K.Feng, C.Taug // IEEE Trans. Inform.Theory, 2017. – Vol.63, № 1. – P. 167-176.

16. Домашнев А.В. Программирование алгоритмов защиты информации / А.В. Домашнев, М.М. Грунтович, В.О. Попов, Д.И. Правиков, А.Ю. Щербаков, И.В. Прокофьев. – М.: Нолидж. – 2002. – 409 с.

17. Б. Шнайдер Прикладная криптография. Протоколы, алгоритмы, исходные тексты на языке Си / Б. Шнайдер – М.:Изд-во ТРИУМФ, 2002. – 816 с.

18. Венбо Мао. Современная криптография. Теория и практика / Венбо Мао; [пер. с англ.]. – М.:Изд. Дом «Вильямс», 2005. – 786 с.

19. ДСТУ 3396.0-96. Захист інформації. Технічний захист інформації. Основні положення. – К.: Держстандарт України. – 1997. – 14 с.

20. ДСТУ 3396.2-96. Захист інформації. Технічний захист інформації. Терміни та визначення. – К.: Держстандарт України. – 1997. – 11 с.

21. Конхейм А. Г. Основы криптографии / Конхейм А. Г. – М.:Радио и Связь, 1987. – 412 с.

22. Математичні основи криптографії / [Кузнецов Г. В., Фомічов В. В., Сушко С. О., Фомічова Л. Я.]. – Дніпропетровськ:НГУ, 2004. – 389 с.

23. Аналіз властивостей алгоритмів блокового симетричного шифрування (за результатами міжнародного проекту NESSIE): Радіотехніка. Всеукраїнський міжвідомчий науково-технічний збірник, вип. 141 / [Горбенко І.Д., Гулак Г.М., Олійников Р.В. та ін.]. – Харків:ХНУРЕ, 2005. – С.7-24.

24. Задірака В.К. Комп’ютерна криптологія: Підручник / В.К. Задірака, О.С. Олексюк. – К.: Вища школа. – 2002. – 504с.

25. Задірака В.К. Комп’ютерна арифметика багато розрядних чисел: Наукове видання / В.К. Задірака, О.С. Олексик. – К.:Вища школа. – 2003. – 264 с.

26. Иванов М.А. Криптографические методы защиты информации в компьютерных системах и сетях / М.А. Иванов. – М.:Кудиц-Образ. – 2001. – 368 с.

27. Фомичев В.М. Дискретная математика и криптология / В.М. Фомичев. – М.:Диалог-МИФИ. – 2003. – 379 с.

28. Харин Ю.С. Математические и компьютерные основы криптологии / Ю.С. Харин, В.И. Берник, Г.В. Матвеев, С.В. Агиевич. – Мн.: Новое знание. – 2003. – 382 с.

29. Boroujerdi N. Cloud Computing: Changing Cogitation about Computing / N. Boroujerdi, S. Nazem // IJCSI International Journal of Computer Science Issues. – Vol. 9. – Issue 4. – 2012. – №3. – PP. 169-180.

30. Bos J.N. Additional chain heuristics / J.N. Bos, M. Coster // Cryptographic Hardware and Embedded System – CHES'2014. LNCS-2116, Springer-Verlag. – 2014. – P.143-151.

31. Панкратова И.А. Булевы функции в криптографии: учеб. Пособие / И.А. Панкратова. – Томск:Издательский Дом Томского государственного университета. – 2014. – 88 с.

32. Dalai D. K. On some necessary conditions of Boolean functions to resist algebraic attack / D.K. Dalai // Ph. D. Thesis. Kolkata, India, – 2006.

33. Meier W. Algebraic attack and decomposition of Boolean functions / W. Meier, E. Pasalic, and C. Carlet // LNCS. – 2004. – Vol. 3027. – P.474–491.

34. Du Y. On the Resistance of Boolean Functions against Fast Algebraic Attacks / Y. Du, F. Zhang, M. Liu // Information Security and Cryptology. – ICISC 2011. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg. – 2012. – Vol. 7259. – PP.261-274.

35. Salagean A. Estimating the nonlinearity of Boolean functions using probabilistic linearity tests / A. Salagean A. // Loughborough University. Conference contribution. Stanica, Pantelimon, – 2020.

36. Carlet C. Boolean Functions for Cryptography and Coding Theory / C. Carlet // Cambridge: Cambridge University Press. – 2021. – P.574.

37. Behera P., Gangopadhyay S. An improved hybrid genetic algorithm to construct balanced Boolean function with optimal cryptographic properties / P. Behera, S. Gangopadhyay // Evolutionary Intelligence, Springer. – 2021. – P.1-15.

38. Doroshenko A. Acceleration of boolean transformations nonlinearity testing for cryptographic algorithms / Anna Doroshenko, Oleksandr Markovskyi // International Conference ICSFTI2019 (Kyiv, May 14−15, 2019). Kyiv, 2019. – P. 35-40.

39. Rusanova O. Energy-aware task scheduling algorithm for mobile computing / Olga Rusanova, Igor Boyarshin, Anna Doroshenko // International Conference ICSFTI2020 (Kyiv, May 13, June 15, 20120). Kyiv, 2020. – P. 107-113.

40. Дорошенко А. Ю. Метод прискореного тестування нелінійності булевих перетворень криптографічних алгоритмів / А.Ю. Дорошенко, В.Ю. Куц // Матеріали II міжнарод. наук.-практ. конф. Наука та концепції: (м. Київ, 29-30 квіт. 2019 р.). Київ, 2019. – С. 15-18.

41. Boyarshin I. Request balancing method for increasing their processing efficiency with information duplication in a distributed data storage system / I. Boyarshin, A. Doroshenko, P. Rehida // Technical sciences and technologies. – 2021. – № 2 (26).

42. Stallings W. Cryptography and Network Security: Principles and Practice / William Stallings // Prentice Hall. – 2006. – P. 680.

43. Cozzens M. The Mathematics of Encryption: An Elementary Introduction / Margaret Cozzens, Steven J. Miller // American Mathematical Soc. – 2013. – P. 332.

44. Easttom W. Modern Cryptography: Applied Mathematics for Encryption and Information Security / William Easttom // Springer Nature. – 2020. – P. 390.

45. Crama Y. Boolean Functions: Theory, Algorithms, and Applications / Yves Crama, Peter L. Hammer // Cambridge University Press. – 2011. – P. 710.

46. Tokareva N. Bent Functions: Results and Applications to Cryptography / Natalia Tokareva // Academic Press. – 2015. – P. 220.

47. Di Pietro R. Cryptographic Strength Evaluation of Key Schedule Algorithms / Roberto Di Pietro, Shazia Afzal, Muhammad Yousaf, Humaira Afzal, Nawaf Alharbe, Muhammad Rafiq Mufti // Security and Communication Networks, Hindawi. – 2020.

48. Doğanaksoy A. Cryptographic Randomness Testing of Block Ciphers and Hash Functions / Ali Doğanaksoy, Baris Ege, Onur Koçak, Fatih Sulak // IACR Cryptology ePrint Archive. – 2010. – P. 564.

49. Mouha N. Extending NIST’s CAVP Testing of Cryptographic Hash Function Implementations / Nicky Mouha, Christopher Celi. – 2020.

50. Яремчук Ю .Є. Аналіз статистичної безпеки криптоалгоритмів асиметричного шифрування на основі рекурентних послідовностей / Ю.Є. Яремчук // Інформатика та математичні методи в моделюванні. – 2014. – Том 4. – № 4. – С. 357-362.

51. Mengdi Z. Overview of Randomness Test on Cryptographic Algorithms // Zhang Mengdi, Zhang Xiaojuan, Zhu Yayun, Miao Siwei // Journal of Physics: Conference Series. – 2021 – Vol. 1861 (1). – P. 012009.

52. Alrubaiei M. Critical Analysis of Cryptographic Algorithms / Mazoon Alrubaiei, Thuraiya AlYarubi, Basant Kumar // 2020 8th International Symposium on Digital Forensics and Security (ISDFS). – 2020. – P. 1-7.

53. Andraşiu M. Evaluation of Cryptographic Algorithms // Mircea Andraşiu, Emil Simion // Romanian Economic Business Review. – 2011. – Vol. 5. – P. 52-62.

54. Jorstad N. Cryptographic algorithm metrics / Norman Jorstad, Landgrave Smith. – 2021.

Додатки

Додаток А

Лістинг програми

main.rs

```
#![allow(unused)]
use rand::Rng;
use std::thread;
use std::ops::Index;
use std::ops::IndexMut;
use std::sync::{Arc, Mutex};

const PARAMETERS: u32 = 3;

fn main() {
    const STATISTIC_COUNT: u32 = 10;
    const THREAD_COUNT: usize = 2;

    let mut passes_stats: Vec<u32> = Vec::with_capacity(STATISTIC_COUNT as usize);
    let mut percent_stats = Vec::with_capacity(STATISTIC_COUNT as usize);

    let mutex = Arc::new(Mutex::new((percent_stats, passes_stats)));
    let mut threads = Vec::new();
    for thread_id in 0..THREAD_COUNT {
        let mutex_clone = Arc::clone(&mutex);
         threads.push(thread::spawn(move || {
            println!("inside {}", thread_id);
            let mut random = FastStdRand::new();
            for i in (thread_id..STATISTIC_COUNT as usize).step_by(THREAD_COUNT) {
                println!("------------------- {} ---------------------", i);
                let test_nonlinear_function = Function::generate_nonlinear_func(&mut
random);
                let real_nl = test_nonlinear_function.calculate_nonlinearity();
                let sorted_stats = test_nonlinear_function.get_sorted_statistics();
                let (curr_n, curr_nonlin) =
test_nonlinear_function.calculate_min_h(sorted_stats);
                let percent = curr_nonlin as f32 / real_nl as f32;
                println!("Real = {} and curr = {} (percent = {})", real_nl, curr_nonlin,
percent);
                let mut data = mutex_clone.lock().unwrap();
                // data.0 == percent_stats
                // data.1 == passes_stats
                data.0.push(percent);
                data.1.push(curr_n);
                drop(data);
            }
        }))
    }
    threads.into_iter().for_each(move |c| { let _ = c.join().unwrap(); });

    let mut data = mutex.lock().unwrap();

    // Stats
    let mut sum = 0;
```

```
    let mut sum_percent = 0.0;
    let mut max_percent = 0.0;
    for k in 0..STATISTIC_COUNT as usize{
        // data.0 == percent_stats
        // data.1 == passes_stats
        sum += data.1[k];
        let percent = data.0[k];
        sum_percent += percent;
        println!("percent = {}", percent);
        if percent > max_percent { max_percent = percent }
    }
    let mean = (sum / STATISTIC_COUNT) as u32 + 1;
    let mean_percent = sum_percent as f32 / STATISTIC_COUNT as f32;
    println!("mean = {}", mean);
    println!("percent mean = {}", mean_percent);
    println!("max percent = {}", max_percent);
}

struct FastStdRand {
    buffer: u32,
    roof: u32,
    index: u32,
}

impl FastStdRand {
    fn new() -> FastStdRand {
        let value = 1 << (std::mem::size_of::<u32>() * 8 - 1);
        FastStdRand{buffer: 0, roof: value, index: value}
    }

    fn gen(&mut self) -> bool {
        if self.index == self.roof {
            self.index = 0;
            self.buffer = rand::thread_rng().gen_range(0..self.roof);
        }
        let result = (self.buffer & 1) == 1;
        self.index += 1;
        self.buffer >>= 1;

        result
    }
}

fn rows_count(parameters: u32) -> usize {
    1usize << parameters
}

fn lambda_functions_count(parameters: u32)  -> usize {
    (rows_count(parameters) << 1) - 2
}

struct Function {
    values: Vec<bool>,
}

// LinearFunction format is x(0),x(1),..x(n) [1,x(0),x(1),..x(n)]
struct LinearFunction {
```

```
    indices: Vec<u32>,
}

impl LinearFunction {
    fn new(mut number: u64) -> LinearFunction {
        let mut indices = Vec::new();
        let mut index = 0;
        while number != 0 {
            if (number & 1) == 1 {
                indices.push(index);
            }
            number >>= 1;
            index += 1;
        }

        LinearFunction{indices}
    }

    fn len(&self) -> usize {
        self.indices.len()
    }
}

impl Index<usize> for LinearFunction {
    type Output = u32;
    fn index<'a>(&'a self, i: usize) -> &'a u32 {
        &self.indices[i]
    }
}

impl IndexMut<usize> for LinearFunction {
    fn index_mut<'a>(&'a mut self, i: usize) -> &'a mut u32 {
        &mut self.indices[i]
    }
}

// Format of a tuple is {number1, number2}
// numbers differ by one bit (for example, {100, 101})
struct Tuple {
    number1: u64,
    number2: u64,
}

impl Tuple {
    fn empty() -> Tuple {
        Tuple{number1: 0, number2: 0}
    }

    fn new(number1: u64, number2: u64) -> Tuple {
        Tuple{number1, number2}
    }
}

impl Index<usize> for Tuple {
    type Output = u64;
    fn index<'a>(&'a self, i: usize) -> &'a u64 {
        assert!(i <= 1);
```

```
        if i == 0 { &self.number1 } else { &self.number2 }
    }
}

impl IndexMut<usize> for Tuple {
    fn index_mut<'a>(&'a mut self, i: usize) -> &'a mut u64 {
        assert!(i <= 1);
        if i == 0 { &mut self.number1 } else { &mut self.number2 }
    }
}

#[derive(Clone, Copy)]
struct Brace {
    elem: (u32, bool),
}

impl Brace {
    fn empty() -> Brace {
        Brace{elem: (0, false)}
    }

    fn new(arg: u32, is_inverted: bool) -> Brace {
        Brace{elem: (arg, is_inverted)}
    }

    fn index(&self) -> u32 {
        self.elem.0
    }

    fn inverted(&self) -> bool {
        self.elem.1
    }
}

impl Function {
    fn new(values: Vec<bool>) -> Function {
        println!("------------------------------------------------------------");
        println!("------------------------------------------------------------");
        Function{values}
    }

    // Calculation of the Hamming distances between the input function
    // and each of lambda functions. Then finding min distance.
    // Also the closest linear functions can be shown
    fn calculate_nonlinearity(&self) -> u32 {
        let mut nonlinearity = u32::MAX;
        let mut array_of_lambda_functions: Vec<Vec<bool>> =
Vec::with_capacity(rows_count(PARAMETERS));
        for i in 2..(lambda_functions_count(PARAMETERS) + 2) {
            let current_lamda_function = Function::get_lambda_function(i as u64);
            let current_nonlinearity = {
                let mut current_nonlinearity = 0;
                for j in 0..current_lamda_function.len() {
                    if self.values[j] != current_lamda_function[j] {
                        current_nonlinearity += 1;
                    }
                }
```

```
                current_nonlinearity
            };

            if current_nonlinearity < nonlinearity {
                nonlinearity = current_nonlinearity;
                array_of_lambda_functions.clear();
            }
            if current_nonlinearity == nonlinearity {
                array_of_lambda_functions.push(current_lamda_function);
            }
        }

        println!("Nonlinearity = {} ", nonlinearity);
        println!("-------------------------------------------------------");

        for min_lambda_function in array_of_lambda_functions.iter() {
            // println!("{:?}", min_lambda_function);
            Function::get_linear_function(min_lambda_function);
        }

        nonlinearity
    }

    // Calculates Hamming distance between the input functions
    // (and Truth Table representation of linear function can be printed)
    // Be careful about input: [x(0), x(1), .. x(n), 1]
    // For example, f=x3 => 0...00100, so function expects 4
    fn calculate_h(&self, indices: &Vec<u32>) -> u32 {
        let mut dec_form = 0;
        for index in indices.iter() {
            dec_form |= 1 << (PARAMETERS - index);
        }
        let lamda_function = Function::get_lambda_function(dec_form);

        // For Truth Table representation uncomment next line
        // println!("{:?}", lamda_function);

        let mut nonlinearity = 0;
        for j in 0..lamda_function.len() {
            if self.values[j] != lamda_function[j] { nonlinearity += 1; }
        }
        if nonlinearity > (rows_count(PARAMETERS) as u32 >> 1) {
            nonlinearity = rows_count(PARAMETERS) as u32 - nonlinearity;
        }

        // println!(" H = {} ", nonlinearity);

        nonlinearity
    }

    fn calculate_min_h(&self, sorted_stats: [(u32, f64); PARAMETERS as usize]) -> (u32,
u32) {
        let mut sorted: [u32; PARAMETERS as usize] = [0; PARAMETERS as usize];
        let mut distances: Vec<u32> = Vec::with_capacity(PARAMETERS as usize);
        for i in 0..sorted_stats.len() {
            sorted[i] = sorted_stats[i].0;
        }
```

```
    print!("sorted = [ ");
    for s in sorted.iter() {
        print!("x{} ", s);
    }
    println!("]");
    println!("============================================================");

    let mut selected = Vec::new();
    selected.push(vec![sorted[0]]);
    let mut min_h = self.calculate_h(&selected[0]);
    distances.push(min_h);

    let mut passes = 1;
    for k in 1..sorted_stats.len() {
        let mut a = 0;
        let mut h = 0;
        while h < selected.len() - a {
            let h1 = distances[h];
            let mut sel = selected[h].clone();
            sel.push(sorted[k]);
            let h2 = self.calculate_h(&sel);
            if h2 < min_h { min_h = h2; }
            passes += 1;
            if h2 <= h1 {
                selected[h] = sel;
                distances[h] = h2;
            } else if !Function::already_added(&selected, sorted[k]) {
                selected.push(vec![sorted[k]]);
                distances.push(self.calculate_h(selected.last().unwrap()));
                passes += 1;
                a +=1;
            }
            h += 1;
        }
    }
    println!("amount of args = {}", PARAMETERS);
    println!("nonlinearity = {}", min_h);
    let mut real_passes = PARAMETERS + passes;
    println!("passes = {}", real_passes);

    (real_passes, min_h)
}

fn already_added(selected: &Vec<Vec<u32>>, test: u32) -> bool {
    for sel in selected.iter() {
        if (sel.len() == 1) && (sel[0] == test) { return true; }
    }

    return false;
}

fn get_lambda_function(index: u64) -> Vec<bool> {
    let linear_function = LinearFunction::new(index);
    let mut lambda_function = vec![false; rows_count(PARAMETERS)];
    for i in 0..rows_count(PARAMETERS) {
        let mut current_result = false;
        let mut j = 0;
```

```
            if linear_function[0] == 0 {
                current_result = true;
                j += 1;
            }
            while j < linear_function.len() {
                current_result ^= ((i >> (linear_function[j] - 1)) & 1) > 0;
                j += 1;
            }
            lambda_function[i] = current_result;
        }

        lambda_function
    }

    // Base number is a number, from which we get inverted number by one exact x.
    // Base number and inverted number form tuple.
    // For example, 000 - base number, 001 - number inverted by x2
    fn get_base_number(parameter: u32, index: u32) -> u64 {
        let chunk_size = 1u64 << parameter;
        let chunk_index = index as u64 / chunk_size;
        let chunk_shift = index as u64 % chunk_size;

        chunk_index * (chunk_size << 1) + chunk_shift
    }

    fn get_tuple(base_number: u64, bit_to_invert: u32) -> Tuple {
        Tuple::new(base_number, base_number ^ (1 << bit_to_invert))
    }

    fn get_sorted_statistics(&self) -> [(u32, f64); PARAMETERS as usize] {
        let mut statistic_array = [(0, 0.0); PARAMETERS as usize];
        for i in 0..PARAMETERS {
            let mut count = 0;
            for j in 0..(rows_count(PARAMETERS) >> 1) {
                let current_tuple = Function::get_tuple(Function::get_base_number(i, j as
u32), i);
                if self.values[current_tuple[0] as usize] != self.values[current_tuple[1]
as usize] {
                    count += 1;
                }
            }
            statistic_array[i as usize] = (i as u32, count as f64 /
(rows_count(PARAMETERS) >> 1) as f64);
        }

        statistic_array.sort_by(|left, right| right.1.partial_cmp(&left.1).unwrap());

        for stat in statistic_array.iter() {
            println!("x{} = {}", stat.0, stat.1);
        }
        println!("------------------------------------------------------------");

        statistic_array

        // To get statistics from variables, brobabilities of which are greater than 0.3
        // let mut statistic_transformed_array = Vec::new();
        // for i in 0..statistic_array.len() {
```

```
        //     let statistic = statistic_array[i];
        //     if statistic.1 > 0.3 {
        //         statistic_transformed_array.push(statistic_array[i]);
        //     }
        // }
        //
        // for stat in statistic_transformed_array.iter() {
        //     println!("x{} = {}", stat.0, stat.1);
        // }
        // println!("------------------------------------------------------------");
        //
        // statistic_transformed_array
    }

    fn generate_nonlinear_func(random: &mut FastStdRand) -> Function {
        let mut result = vec![false; rows_count(PARAMETERS)];
        let mut amount_of_ones = 0;
        for i in 0..rows_count(PARAMETERS) {
            let new_bit = random.gen();
            result[i] = new_bit;
            if new_bit { amount_of_ones += 1; }
        }

        // Check whether generated function is balanced.
        // Need a correction if not balanced
        let new_bit = amount_of_ones < (rows_count(PARAMETERS) >> 1);
        while amount_of_ones != (rows_count(PARAMETERS) >> 1) {
            let mut pos = rand::thread_rng().gen_range(0..rows_count(PARAMETERS));
            while result[pos] == new_bit && ((pos + 1) < rows_count(PARAMETERS)) {
                pos += 1;
            } // very clever stuff
            pos += 1;
            if result[pos - 1] != new_bit { // if what we're looking for
                result[pos - 1] = new_bit;
                if new_bit {
                    amount_of_ones += 1;
                } else {
                    amount_of_ones -= 1;
                }
            } // else we've stopped because pos == rows_count => bad luck => try
everything again
        }

        Function{values: result}
    }

    // method for finding linear function from lamda function
    fn get_linear_function(current_lambda_func: &Vec<bool>) {
        let mut presence_table = vec![false; rows_count(PARAMETERS)];

        for row in 0..rows_count(PARAMETERS) {
            if !current_lambda_func[row] { continue }
            let mut current_arg = [Brace::empty(); PARAMETERS as usize];
            for j in (0..=(PARAMETERS - 1)).rev() {
                let bit = ((row >> j) & 1) == 0;
                current_arg[(PARAMETERS - j - 1) as usize] = Brace::new((PARAMETERS - j -
1), bit);
```

```
            }

            // processing of current_arg
            for combination in 0..rows_count(PARAMETERS) {
                let mut arguments = Vec::new();
                let mut xor_zero = false;
                for b in 0..PARAMETERS {
                    let current_brace_bit = ((combination >> (PARAMETERS - 1 - b)) & 1) ==
0;
                    if !current_brace_bit {
                        arguments.push(current_arg[b as usize].index());
                    } else {
                        // current_brace_bit == 1 => take second place inside brace
                        if current_arg[b as usize].inverted() { // +1
                            continue; // don't put this x, try next if any left
                        } else { // +0
                            xor_zero = true;
                            break;
                        }
                    }
                }
                if xor_zero { continue }
                let mut current_index = 0;
                for argument in arguments {
                    current_index += (1u64 << argument);
                }
                presence_table[current_index as usize] = !presence_table[current_index as
usize];
            }
        }

        // analize presence_table and show the result
        let mut not_first = false;
        for index in (0..=(rows_count(PARAMETERS)-1)).rev() {
            if presence_table[index] {
                if not_first {
                    print!(" + ");
                } else {
                    not_first = true;
                }
                if index == 0 {
                    print!("1");
                } else {
                    let mut right_zeros = 0;
                    for param in 0..PARAMETERS {
                        if ((index >> param) & 1) == 1 { break }
                        right_zeros += 1;
                    }
                    print!("x{}", right_zeros);
                }
            }
        }
        println!();
        println!("------------------------------------------------------------");
    }
}
```

```
impl Index<usize> for Function {
    type Output = bool;
    fn index<'a>(&'a self, i: usize) -> &'a bool {
        &self.values[i]
    }
}
```